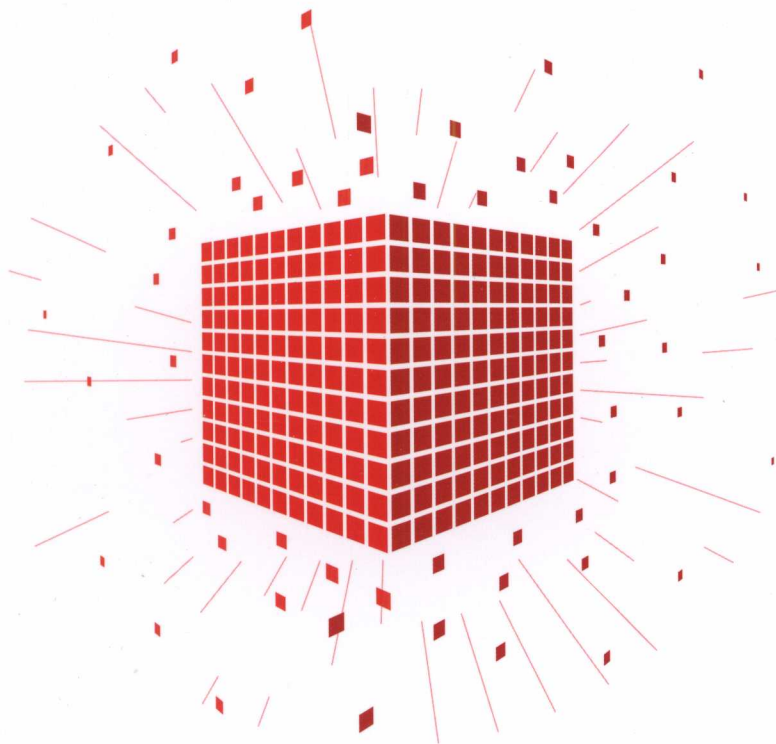


版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



An In-Depth Guide to Hadoop HDFS

深度剖析 Hadoop HDFS

林意群◎编著



机械工业出版社
China Machine Press

内容简介

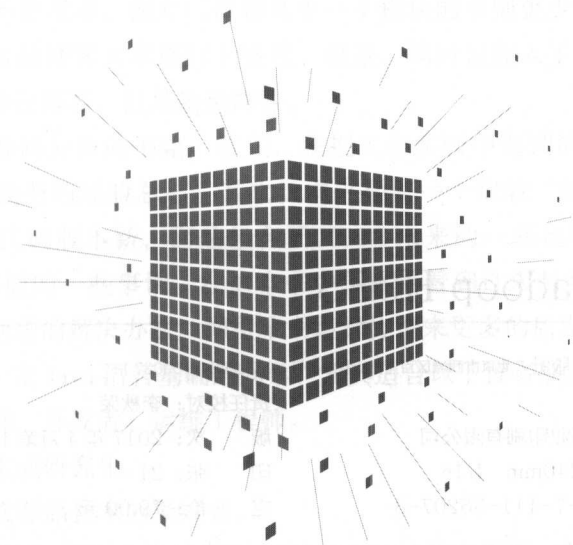
本书基于Hadoop 2.7.1版本进行分析，全面描述了HDFS 2.X的核心技术与解决方案。书中描述了HDFS内存存储、异构存储等几大核心设计，包括源码细节层面的分析，对于HDFS的几个主要使用场景也做了细粒度的分析。还分享了作者在实际应用中的解决方案及扩展思路。阅读本书可以帮助读者从架构设计与功能实现角度了解HDFS 2.X，同时还能学习HDFS 2.X框架中优秀的设计思想、设计模式、Java语言技巧等。这些对于读者全面提高自己分布式技术水平有很大的帮助。本书分为三大部分：核心设计篇、细节实现篇、解决方案篇，“核心设计篇”包括HDFS的数据存储原理、HDFS的数据管理与策略选择机制、HDFS的新颖功能特性；“细节实现篇”包括HDFS的块处理、流量处理等细节，以及部分结构分析；“解决方案篇”包括HDFS的数据管理、HDFS的数据读写、HDFS的异常场景等。本书适合于云计算相关领域研发人员、云计算相关运维工程师、高年级本科生或研究生、热衷于分布式计算研究的人。



技术丛书

深度剖析 Hadoop HDFS

林意群◎编著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

深度剖析 Hadoop HDFS / 林意群编著. —北京: 机械工业出版社, 2017.3
(大数据技术丛书)

ISBN 978-7-111-56207-8

I. 深… II. 林… III. 数据处理软件 IV. TP274

中国版本图书馆 CIP 数据核字 (2017) 第 040479 号

深度剖析 Hadoop HDFS

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 吴 怡

责任校对: 李秋荣

印 刷: 北京诚信伟业印刷有限公司

版 次: 2017 年 4 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 21

书 号: ISBN 978-7-111-56207-8

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光/邹晓东

Preface 前言

我上大学时，就开始在 CSDN 上写技术博客，目的在于记录平时遇到的一些问题以及研究的技术细节，好在将来可以进行查阅。随着时间的增长，我开始专注于某个技术模块，因为这样可以让我对具体某项技术有更深入的研究，写出的内容也会更加系统化，而 HDFS 就是其中一个我持续研究的技术模块。同时作为一名 Hadoop 社区的活跃贡献者，我也会将社区上一些比较有意思的东西分享到博客上，许多博友给了不少反馈，描述他们在工作中碰到的一些实际问题。在这样不断的写作、交流过程中，我得到了快速成长。目前大数据领域相关的书籍并不是很多，而专门讲解其中一个模块的书则更少，所以我将我过去一年多时间内关于 HDFS 的博客文章进行了整理、改进，同时也加入了一些新的内容。可以说，本书的内容源自博客，但是超越博客。

本书不会是纯源码分析的书籍。首先，我把工作实践中遇到的许多经验写入了书中，第 7 章便属于纯实践型的经验总结。其次，本书会是一个比较“新”的书，这里的“新”并不是指所分析的代码版本新，而是包含了 HDFS 未来的一些比较棒的功能特性，以及 Hadoop 社区目前在做的一些事情。在这本书中，你会看到许多与社区相关的 JIRA，了解如何从社区上找到问题的解决办法。期待本书能给你带来更多的启发。

本书适合具有一定 Java 语言基础的同学，尤其适合以下读者朋友：

- 大数据架构师、开发者、运维工程师。
- 高年级本科生或研究生。
- 热衷于分布式存储技术的爱好者。

本书分为三大部分，“核心设计篇”介绍 HDFS 的基本原理、数据管理与策略等，“细节实现篇”介绍 HDFS 的块处理、流量处理、结构分析等，“解决方案篇”介绍数据管理技术与方案、数据读写技术、异常处理等。

第一部分“核心设计篇”包括内容如下：

第 1 章介绍 HDFS 现有的数据存储方式，主要介绍其中的内存存储和异构存储两个方面。

第2章介绍 HDFS 目前内部几种主要的功能机制，包括缓存管理、快照管理等。

第3章介绍 HDFS 比较新颖的一些功能，以及目前较少被人用到的功能特性。

第二部分“细节实现篇”包括内容如下：

第4章介绍 HDFS 的块处理相关操作，主要处理场景包括块如何组织、上报处理的过程以及多余块的清除。

第5章介绍 HDFS 的流量处理过程，包括 HDFS 目前流量处理的场景以及 Balancer 工具的数据平衡原理和优化。

第6章介绍 HDFS 一些特殊的结构对象类，包括这些类的作用、原理以及运用场景。

第三部分“解决方案篇”包括内容如下：

第7章介绍与 HDFS 相关的多套运维管理的操作方案，包括数据迁移、数据监控等方面。

第8章介绍 HDFS 写磁盘时的一些优化策略和改造方案。

第9章介绍 HDFS 的一些异常场景，并给出了相应的解决方案。

由于笔者水平有限，本书难免会有出错或者介绍不明确的地方，恳请读者批评指正，可以发送关于本书的意见和建议到我的个人邮箱：yqtin@apache.org。本书所涉及的源码，大家可以从 Hadoop 的 Git 地址上进行下载：<https://github.com/apache/hadoop>，其中，不同的分支对应不同版本的代码。相关 Git 地址和 CSDN 博客地址如下：

□ Git 地址：<https://github.com/linyiquan>

□ CSDN 地址：<http://blog.csdn.net/androidlushangderen>

感谢机械工业出版社的吴怡编辑，在我写作的过程中，不断指出其中的不足之处，督促和引导我完成本书的编写。

感谢蘑菇街数据平台部的同事们，在工作中不断地给予我帮助和支持，协助我解决各种各样的问题，于是才有了本书中所展现的精彩内容。

林意群

2017年2月

Contents 目 录

前言

第一部分 核心设计篇

第 1 章 HDFS 的数据存储	2
1.1 HDFS 内存存储	2
1.1.1 HDFS 内存存储原理	2
1.1.2 Linux 虚拟内存盘	4
1.1.3 HDFS 的内存存储流程分析	4
1.1.4 LAZY_PERSIST 内存存储的使用	14
1.2 HDFS 异构存储	15
1.2.1 异构存储类型	16
1.2.2 异构存储原理	17
1.2.3 块存储类型选择策略	22
1.2.4 块存储策略集合	24
1.2.5 块存储策略的调用	27
1.2.6 HDFS 异构存储策略的不足之处	28
1.2.7 HDFS 存储策略的使用	30
1.3 小结	31

第 2 章 HDFS 的数据管理与策略选择	32
2.1 HDFS 缓存与缓存块	32
2.1.1 HDFS 物理层面缓存块	33
2.1.2 缓存块的生命周期状态	34
2.1.3 CacheBlock、UnCacheBlock 场景触发	36
2.1.4 CacheBlock、UnCacheBlock 缓存块的确定	38
2.1.5 系统持有的缓存块列表如何更新	39
2.1.6 缓存块的使用	40
2.1.7 HDFS 缓存相关配置	40
2.2 HDFS 中心缓存管理	42
2.2.1 HDFS 缓存适用场景	43
2.2.2 HDFS 缓存的结构设计	43
2.2.3 HDFS 缓存管理机制分析	45
2.2.4 HDFS 中心缓存疑问点	55
2.2.5 HDFS CacheAdmin 命令使用	56
2.3 HDFS 快照管理	58
2.3.1 快照概念	59
2.3.2 HDFS 中的快照相关命令	59
2.3.3 HDFS 内部的快照管理机制	60
2.3.4 HDFS 的快照使用	71
2.4 HDFS 副本放置策略	72
2.4.1 副本放置策略概念与方法	72
2.4.2 副本放置策略的有效前提	73
2.4.3 默认副本放置策略的分析	73
2.4.4 目标存储好坏的判断	82
2.4.5 chooseTargets 的调用	83
2.4.6 BlockPlacementPolicyWithNodeGroup 继承类	84
2.4.7 副本放置策略的结果验证	85
2.5 HDFS 内部的认证机制	85
2.5.1 BlockToken 认证	85
2.5.2 HDFS 的 Sasl 认证	91
2.5.3 BlockToken 认证与 HDFS 的 Sasl 认证对比	97

2.6	HDFS 内部的磁盘目录服务	98
2.6.1	HDFS 的三大磁盘目录检测扫描服务	98
2.6.2	DiskChecker: 坏盘检测服务	99
2.6.3	DirectoryScanner: 目录扫描服务	104
2.6.4	VolumeScanner: 磁盘目录扫描服务	110
2.7	小结	116
第 3 章	HDFS 的新颖功能特性	117
3.1	HDFS 视图文件系统: ViewFileSystem	117
3.1.1	ViewFileSystem: 视图文件系统	118
3.1.2	ViewFileSystem 内部实现原理	119
3.1.3	ViewFileSystem 的使用	125
3.2	HDFS 的 Web 文件系统: WebHdfsFileSystem	126
3.2.1	WebHdfsFileSystem 的 REST API 操作	127
3.2.2	WebHdfsFileSystem 的流程调用	129
3.2.3	WebHdfsFileSystem 执行器调用	130
3.2.4	WebHDFS 的 OAuth2 认证	133
3.2.5	WebHDFS 的使用	135
3.3	HDFS 数据加密空间: Encryption zone	136
3.3.1	Encryption zone 原理介绍	136
3.3.2	Encryption zone 源码实现	136
3.3.3	Encryption zone 的使用	144
3.4	HDFS 纠删码技术	145
3.4.1	纠删码概念	145
3.4.2	纠删码技术的优劣势	146
3.4.3	Hadoop 纠删码概述	147
3.4.4	纠删码技术在 Hadoop 中的实现	148
3.5	HDFS 对象存储: Ozone	152
3.5.1	Ozone 介绍	153
3.5.2	Ozone 的高层级设计	154
3.5.3	Ozone 的实现细节	157
3.5.4	Ozone 的使用	157
3.6	小结	158

第二部分 细节实现篇

第 4 章 HDFS 的块处理	160
4.1 HDFS 块检查命令 fsck	160
4.1.1 fsck 参数使用	160
4.1.2 fsck 过程调用	161
4.1.3 fsck 原理分析	162
4.1.4 fsck 使用场景	171
4.2 HDFS 如何检测并删除多余副本块	171
4.2.1 多余副本块以及发生的场景	172
4.2.2 OverReplication 多余副本块处理	172
4.2.3 多余副本块清除的场景调用	177
4.3 HDFS 数据块的汇报与处理	179
4.3.1 块处理的五大类型	179
4.3.2 toAdd: 新添加的块	181
4.3.3 toRemove: 待移除的块	184
4.3.4 toInvalidate: 无效的块	186
4.3.5 toCorrupt: 损坏的块	189
4.3.6 toUC: 正在构建中的块	191
4.4 小结	193
第 5 章 HDFS 的流量处理	194
5.1 HDFS 的内部限流	194
5.1.1 数据的限流	194
5.1.2 DataTransferThrottler 限流原理	196
5.1.3 数据流限流在 Hadoop 中的使用	198
5.1.4 Hadoop 限流优化点	202
5.2 数据平衡	204
5.2.1 Balancer 和 Dispatcher	204
5.2.2 数据不平衡现象	207
5.2.3 Balancer 性能优化	207

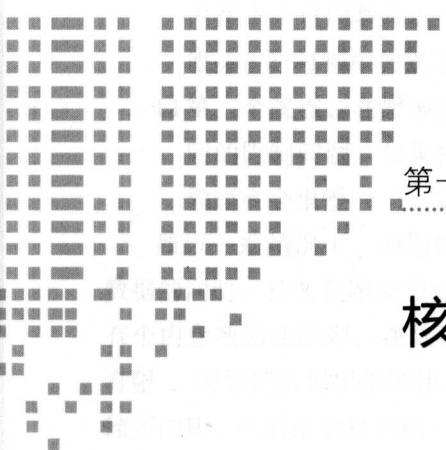
5.3	HDFS 节点内数据平衡	210
5.3.1	磁盘间数据不平衡现象及问题	211
5.3.2	传统的磁盘间数据不平衡解决方案	211
5.3.3	社区解决方案: DiskBalancer	212
5.4	小结	216
第 6 章	HDFS 的部分结构分析	217
6.1	HDFS 镜像文件的解析与反解析	217
6.1.1	HDFS 的 FsImage 镜像文件	218
6.1.2	FsImage 的解析	218
6.1.3	FsImage 的反解析	221
6.1.4	HDFS 镜像文件的解析与反解析命令	226
6.2	DataNode 数据处理中心 DataXceiver	227
6.2.1	DataXceiver 的定义和结构	228
6.2.2	DataXceiver 下游处理方法	232
6.2.3	ShortCircuit	232
6.2.4	DataXceiver 的上游调用	233
6.2.5	DataXceiver 与 DataXceiverServer	234
6.3	HDFS 邻近信息块: BlockInfoContiguous	235
6.3.1	triplets 对象数组	236
6.3.2	BlockInfoContiguous 的链表操作	239
6.3.3	块迭代器 BlockIterator	244
6.4	小结	246

第三部分 解决方案篇

第 7 章	HDFS 的数据管理	248
7.1	HDFS 的读写限流方案	248
7.1.1	限流方案实现要点以及可能造成的影响	248
7.1.2	限流方案实现	249
7.1.3	限流测试结果	250

7.2	HDFS 数据资源使用量分析以及趋势预测	250
7.2.1	要获取哪些数据	251
7.2.2	如何获取这些数据	251
7.2.3	怎么用这些数据	254
7.3	HDFS 数据迁移解决方案	257
7.3.1	数据迁移使用场景	257
7.3.2	数据迁移要素考量	258
7.3.3	HDFS 数据迁移解决方案: DistCp	259
7.3.4	DistCp 优势特性	260
7.3.5	Hadoop DistCp 命令	264
7.3.6	DistCp 解决集群间数据迁移实例	265
7.4	DataNode 迁移方案	265
7.4.1	迁移方案的目标	266
7.4.2	DataNode 更换主机名、ip 地址时的迁移方案	267
7.5	HDFS 集群重命名方案	268
7.6	HDFS 的配置管理方案	271
7.6.1	HDFS 配置管理的问题	271
7.6.2	现有配置管理工具	272
7.6.3	运用 Git 来做配置管理	272
7.7	小结	273
第 8 章	HDFS 的数据读写	274
8.1	DataNode 引用计数磁盘选择策略	274
8.1.1	HDFS 现有磁盘选择策略	274
8.1.2	自定义磁盘选择策略	279
8.2	Hadoop 节点“慢磁盘”监控	282
8.2.1	慢磁盘的定义以及如何发现	282
8.2.2	慢磁盘监控	284
8.3	小结	287
第 9 章	HDFS 的异常场景	288
9.1	DataNode 慢启动问题	288

9.1.1	DataNode 慢启动现象	288
9.1.2	代码追踪分析	290
9.1.3	参数可配置化改造	293
9.2	Hadoop 中止下线操作后大量剩余复制块问题	295
9.2.1	节点下线操作的含义及问题	295
9.2.2	死节点“复活”	297
9.2.3	Decommission 下线操作如何运作	299
9.2.4	中止下线操作后移除残余副本块解决方案	303
9.3	DFSOutputStream 的 DataStreamer 线程泄漏问题	306
9.3.1	DFSOutputStream 写数据过程及周边相关类、变量	306
9.3.2	DataStreamer 数据流对象	307
9.3.3	ResponseProcessor 回复获取类	311
9.3.4	DataStreamer 与 DFSOutputStream 的关系	313
9.3.5	Streamer 线程泄漏问题	316
9.4	小结	319
附录	如何向开源社区提交自己的代码	320



第一部分 *Part 1*

核心设计篇

HDFS 的数据存储

本章将从 HDFS 的数据存储开始说起，因为正是先有了数据的存储，才有后续的写入和管理等操作。HDFS 的数据存储包括两块：一块是 HDFS 内存存储，另一块是 HDFS 异构存储。HDFS 内存存储是一种十分特殊的存储方式，将会对集群数据的读写带来不小的性能提升，而 HDFS 异构存储则能帮助我们更加合理地把数据存到应该存的地方。

1.1 HDFS 内存存储

HDFS 的内存存储是 HDFS 所有数据存储方式中比较特殊的一种，与之后将会提到的 HDFS 缓存有一些相同之处：都用机器的内存作为存储数据的载体。不同之处在于：HDFS 缓存需要用户主动设置目标待缓存的文件、目录，其间需要使用 HDFS 缓存管理命令。而 HDFS 内存存储策略：LAZY_PERSIST 则直接将内存作为数据存放的载体，可以这么理解，此时节点的内存也充当了一块“磁盘”。只要将文件设置为内存存储方式，最终会将其存储在节点的内存中。综合地看，HDFS 缓存更像是改进用户使用的一种功能，而 HDFS 内存存储则是从底层扩展了 HDFS 的数据存储方式。本节将对 HDFS 内存存储策略进行更细致的分析。

1.1.1 HDFS 内存存储原理

对于内存存储的存储策略，可能很多人会存有这么几种看法：

□ 数据临时维持在内存中，服务一停止，数据全部丢失。

□ 数据存在于内存中，在服务停止时做持久化处理，最终将数据全部写入到磁盘。

仔细来看以上这 2 种观点，其实都有不小的瑕疵：

□ 第一个观点，服务一旦停止，内存数据全部丢失，这是无法接受的，我们只能容忍内存中少量的数据丢失。这个观点的另一个问题是，内存的存储空间是有限的，在服务运行过程中如果不及时处理一部分数据，内存空间迟早会被耗尽。

□ 第二个观点，在服务停止退出的时候做持久化操作，同样会面临上面提到的内存空间的限制问题。如果机器的内存足够大，数据可能会很多，那么最后写入磁盘的阶段速度会很慢。

所以一般情况下，通用的、比较好的做法是异步持久化，什么意思呢？在内存存储新数据的同时，持久化距离当前时刻最远（存储时间最早）的数据。换一个通俗的解释，好比有个内存数据块队列，在队列头部不断有新增的数据块插入，就是待存储的块，因为资源有限，需要把队列尾部的块，也就是更早些时间点的块持久化到磁盘中，这样才有空间存储新的块。然后形成这样的一个循环，新的块加入，老的块移除，保证了整体数据的更新。

HDFS 的 LAZY_PERSIST 内存存储策略用的就是这套方法，原理如图 1-1。

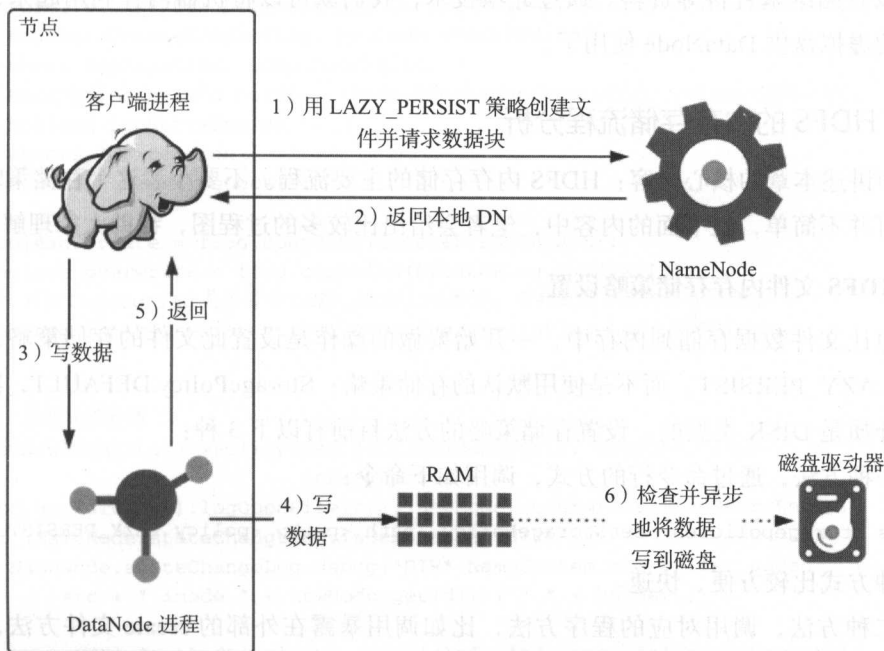


图 1-1 LAZY_PERSIST 策略原理图

上面描述的原理在图中的表示是第 4 个步骤和第 6 个步骤。第 4 步写数据到内存中，第 6 步异步地将数据写到磁盘。前面几个步骤是如何设置 StorageType 的操作，在下文中会

具体提到。所以异步存储的大体步骤可以归纳如下：

- 1) 对目标文件目录设置 StoragePolicy 为 LAZY_PERSIST 的内存存储策略。
- 2) 客户端进程向 NameNode 发起创建 / 写文件的请求。
- 3) 客户端请求到具体的 DataNode 后 DataNode 会把这些数据块写入 RAM 内存中，同时启动异步线程服务将内存数据持久化写到磁盘上。

内存的异步持久化存储是内存存储与其他介质存储不同的地方。这也是 LAZY_PERSIST 名称的缘由，数据不是马上落盘，而是懒惰的、延时地进行处理。

1.1.2 Linux 虚拟内存盘

这里需要了解一个额外的知识点：Linux 虚拟内存盘。之前笔者也一直有个疑惑，内存也可以当作一个块盘使用？内存不就是临时存数据用的吗？于是在学习此模块知识之前，特意查了相关的资料。其实在 Linux 中，的确有将内存模拟为一个块盘的技术，叫**虚拟内存盘 (RAM disk)**。这是一种模拟的盘，实际数据都是存放在内存中的。虚拟内存盘可以在某些特定的内存式存储文件系统下结合使用，比如 tmpfs、ramfs。关于 tmpfs 的具体内容，大家可以查阅维基百科等资料。通过此项技术，我们就可以将机器内存利用起来，作为一块独立的虚拟盘供 DataNode 使用了。

1.1.3 HDFS 的内存存储流程分析

下面讲述本章的核心内容：HDFS 内存存储的主要流程。不要小看这个存储策略，里面的过程可并不简单，在下面的内容中，笔者会给出比较多的过程图，帮助大家理解。

1. HDFS 文件内存存储策略设置

要想让文件数据存储到内存中，一开始要做的操作是设置此文件的存储策略，即上面提到的 LAZY_PERSIST，而不是使用默认的存储策略：StoragePolicy.DEFAULT，默认策略的存储介质是 DISK 类型的。设置存储策略的方法目前有以下 3 种：

第一种方法，通过命令行的方式，调用如下命令：

```
hdfs storagepolicies -setStoragePolicy -path <path> -policy LAZY_PERSIST
```

这种方式比较方便、快速。

第二种方法，调用对应的程序方法，比如调用暴露在外部的 create 文件方法，但是得带上参数 CreateFlag.LAZY_PERSIST。如下所示：

```
FSDataOutputStream fos =
    fs.create(
        path,
        FsPermission.getFileDefault(),
```

```
EnumSet.of(CreateFlag.CREATE, CreateFlag.LAZY_PERSIST),
bufferLength,
replicationFactor,
blockSize,
null);
```

上述方式最终调用的是 DFSClient 的 create 同名方法，如下所示：

```
// DFSClient 创建文件方法
public DFSOutputStream create(String src, FsPermission permission,
    EnumSet<CreateFlag> flag, short replication, long blockSize,
    Progressable progress, int buffersize, ChecksumOpt checksumOpt)
    throws IOException {
    return create(src, permission, flag, true,
        replication, blockSize, progress, buffersize, checksumOpt, null);
}
```

方法经过 RPC 层层调用，经过 FSNamesystem，最终会到 FSDirWriteFileOp 的 startFile 方法，在此方法内部，会有设置存储策略的动作：

```
static HdfsFileStatus startFile(
    FSNamesystem fsn, FSPermissionChecker pc, String src,
    PermissionStatus permissions, String holder, String clientMachine,
    EnumSet<CreateFlag> flag, boolean createParent,
    short replication, long blockSize,
    EncryptionKeyInfo ezInfo, INode.BlocksMapUpdateInfo toRemoveBlocks,
    boolean logRetryEntry)
    throws IOException {
    assert fsn.hasWriteLock();

    boolean create = flag.contains(CreateFlag.CREATE);
    boolean overwrite = flag.contains(CreateFlag.OVERWRITE);
    // 判断 CreateFlag 是否带有 LAZY_PERSIST 标识，来判断是否是内存存储策略
    boolean isLazyPersist = flag.contains(CreateFlag.LAZY_PERSIST);

    ...
    // 在此设置策略
    setNewINodeStoragePolicy(fsd.getBlockManager(), newNode, iip,
        isLazyPersist);
    fsd.getEditLog().logOpenFile(src, newNode, overwrite, logRetryEntry);
    if (NameNode.stateChangeLog.isDebugEnabled()) {
        NameNode.stateChangeLog.debug("DIR* NameSystem.startFile: added " +
            src + " inode " + newNode.getId() + " " + holder);
    }
    return FSDirStatAndListingOp.getFileInfo(fsd, src, false, isRawPath);
}
```

这部分的过程调用见图 1-2。

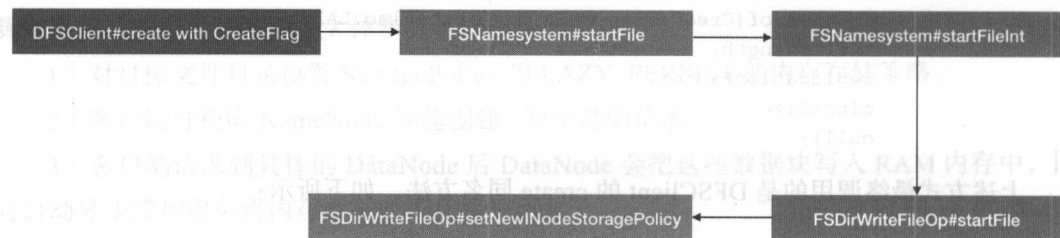


图 1-2 LAZY_PERSIST 策略设置流程图

还有一种方法是通过 FileSystem 的 setStoragePolicy 方法，不过此方法在还未发布的 2.8 版本中提供，如下所示：

```
fs.setStoragePolicy(path, "LAZY_PERSIST");
```

这种方式的优点在于可以用程序动态地设置目标路径的存储方式。

以上就是存储策略的设置过程，这一部分还是非常直接明了的。

2. LAZY_PERSIST 内存存储

当我们为文件设置了 LAZY_PERSIST 的存储方式之后，DataNode 如何进行内存式的存储呢？笔者在下面会分模块、分角色进行介绍。

首先要介绍的是 LAZY_PERSIST 相关结构。在之前的内容中已经提到过，在数据存储的同时会有另外一批数据被异步地持久化，所以这里一定会涉及多个服务对象的合作。这些服务对象的指挥者是 FsDatasetImpl，它是一个管理 DataNode 所有磁盘读写的管家。

在 FsDatasetImpl 中，与内存存储相关的服务对象有 3 个，如图 1-3 所示。

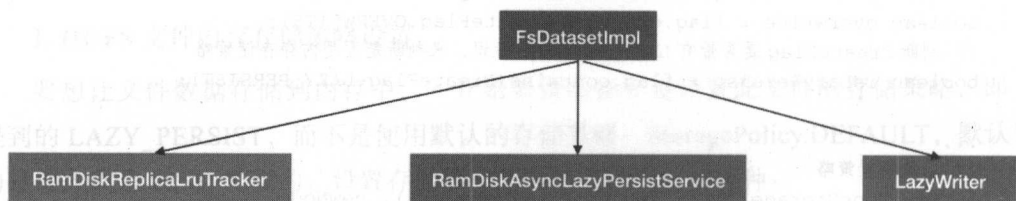


图 1-3 LAZY_PERSIST 相关服务对象

说明如下：

- ❑ RamDiskAsyncLazyPersistService：此对象是异步持久化线程服务，针对每一个磁盘块设置一个对应的线程池，需要持久化到给定磁盘的数据块会被提交到对应的线程池中去。每个线程池的最大线程数为 1。
- ❑ LazyWriter：这是一个线程服务，此线程会不断地从数据块列表中取出数据块，将数据块加入到异步持久化线程池 RamDiskAsyncLazyPersistService 中去执行。

□ **RamDiskReplicaLruTracker**：是副本块跟踪类，此类中维护了所有已持久化、未持久化的副本以及总副本数据信息。所以当副本被最终存储到内存中后，相应地会有副本所属队列信息的变更。当节点内存不足时，会将最近最少被访问的副本块移除。

以上3者的紧密合作，最终实现HDFS的内存存储。下面是具体的角色介绍。

(1) RamDiskReplicaLruTracker

RamDiskReplicaLruTracker起到了一个中间人的角色，它内部维护了多个关系的数据块信息，主要是以下3类：

```
public class RamDiskReplicaLruTracker extends RamDiskReplicaTracker {
    ...
    // blockpool Id 对副本信息的映射图
    Map<String, Map<Long, RamDiskReplicaLru>> replicaMaps;

    // 待写入磁盘的副本队列
    Queue<RamDiskReplicaLru> replicasNotPersisted;

    // 已持久化写入磁盘的映射图
    TreeMultimap<Long, RamDiskReplicaLru> replicasPersisted;
    ...
}
```

这里的 `Queue<RamDiskReplicaLru>` 就是待存入内存存储队列。以上3个变量之间的关系见图1-4。

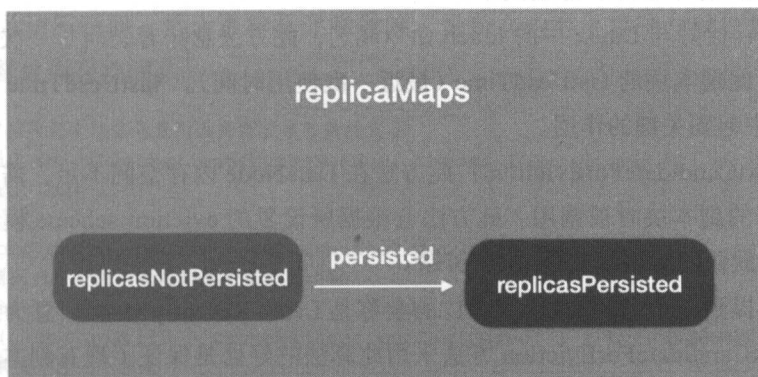


图1-4 RamDisk 副本块结构关系图

RamDiskReplicaLruTracker 中的方法操作绝大多数与这3个变量的增删改动相关，所以逻辑并不复杂，我们只需要了解这些方法有什么作用即可。笔者将方法分成了以下两类：

第一类，异步持久化操作相关方法。如图1-5所示。

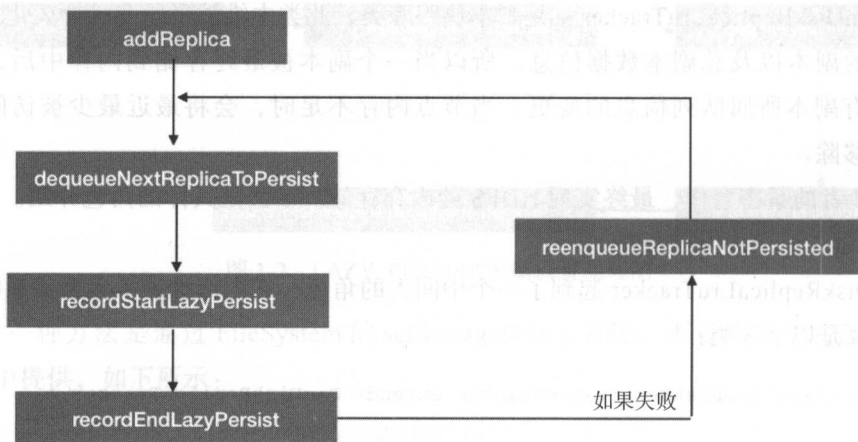


图 1-5 异步持久化操作相关流程图

当节点重启或者有新的文件设置了 LAZY_PERSIST 策略后，就会有新的副本块存储到内存中，同时会加入到 replicaNotPersisted 队列中。经过中间的 dequeueNextReplicaToPersist 方法，取出下一个将被持久化的副本块，进行写磁盘的操作。在持久化的过程中将调用 recordStartLazyPersist、recordEndLazyPersist 这两个方法，标志着持久化状态的变更。

第二类，异步持久化操作无直接关联方法。方法如下：

1) discardReplica：当检测到不再需要某副本的时候（包括副本已被删除，或已损坏的情况），可以从内存中移除、撤销副本。

2) touch：恰好与 Linux 中的 touch 命令同名，此方法意味着访问了一次某特定的副本块，并会更新此副本块的 lastUsedTime（最近一次使用时间）。lastUsedTime 会在后面提到的 LRU 算法中起到关键的作用。

3) getNextCandidateForEviction：此方法在 DataNode 内存空间不足，需要内存额外预留出空间给新的副本块时被调用。此方法会根据所设置的 eviction scheme 模式，选择需要被移除的块，默认的策略模式是 LRU 策略。

这里反复提到一个名词 LRU，LRU 的全称是 Least Recently Used，意为最近最少使用算法。getNextCandidateForEviction 方法采用此算法的好处是保证了现有副本块的一个活跃度，把最近很久没有访问过的块给移除掉。对于这个操作，我们有必要了解其中的细节。

首先 touch 方法会更新副本块最近访问的时间：

```

synchronized void touch(final String bpid,
                        final long blockId) {
    Map<Long, RamDiskReplicaLru> map = replicaMaps.get(bpid);
    RamDiskReplicaLru ramDiskReplicaLru = map.get(blockId);
    ...
}

```

```

// 更新最近访问时间戳，并重新插入数据
if (replicasPersisted.remove(ramDiskReplicaLru.lastUsedTime, ramDiskReplicaLru)) {
    ramDiskReplicaLru.lastUsedTime = Time.monotonicNow();
    replicasPersisted.put(ramDiskReplicaLru.lastUsedTime, ramDiskReplicaLru);
}
}
// 第二步获取候选移除块
synchronized RamDiskReplicaLru getNextCandidateForEviction() {
    // 获取 replicasPersisted 迭代器进行遍历
    final Iterator<RamDiskReplicaLru> it = replicasPersisted.values().iterator();
    while (it.hasNext()) {
        // 因为 replicasPersisted 已经根据时间排好序了，所以取出当前的块进行移除即可
        final RamDiskReplicaLru ramDiskReplicaLru = it.next();
        it.remove();

        Map<Long, RamDiskReplicaLru> replicaMap =
            replicaMaps.get(ramDiskReplicaLru.getBlockPoolId());

        if (replicaMap != null && replicaMap.get(ramDiskReplicaLru.getBlockId()) != null) {
            return ramDiskReplicaLru;
        }

        // 如果副本不存在，则继续下一个副本
    }
    return null;
}

```

这里比较有意思的是，根据已持久化块的访问时间来进行筛选移除，而不是直接在内存块对象中记录访问时间，然后进行排序和移除。最后在内存中移除与候选块属于同一副本信息的块并释放内存空间：

```

// 从内存中移除副本块信息直到满足需要字节数的大小
public void evictBlocks(long bytesNeeded) throws IOException {
    int iterations = 0;

    final long cacheCapacity = cacheManager.getCacheCapacity();
    // 当检测到内存空间不满足外界需要的大小时
    while (iterations++ < MAX_BLOCK_EVICTIONS_PER_ITERATION &&
        (cacheCapacity - cacheManager.getCacheUsed()) < bytesNeeded) {
        // 获取待移除副本信息
        RamDiskReplica replicaState = ramDiskReplicaTracker.getNextCandidate
            ForEviction();

        if (replicaState == null) {
            break;
        }

        if (LOG.isDebugEnabled()) {
            LOG.debug("Evicting block " + replicaState);
        }
    }
}

```

```

...
// 移除内存中的相关块并释放空间
removeOldReplica(replicaInfo, newReplicaInfo, blockFile, metaFile,
    blockFileUsed, metaFileUsed, bpid);
}
}
}

```

(2) LazyWriter

LazyWriter 是一个线程服务，它是一个发动机，循环不断地从队列中取出待持久化的数据块，提交到异步持久化服务中去。其中主要的 run 方法如下所示：

```

public void run() {
    int numSuccessiveFailures = 0;

    while (fsRunning && shouldRun) {
        try {
            // 取出新的副本块并提交到异步服务中，返回是否提交成功的布尔值
            numSuccessiveFailures = saveNextReplica() ? 0 : (numSuccessiveFailures + 1);

            // 如果所有的持久化操作失败，则进行睡眠等待，避免短时间内连续的重试
            if (numSuccessiveFailures >= ramDiskReplicaTracker.numReplicas
                NotPersisted()) {
                Thread.sleep(checkpointerInterval * 1000);
                numSuccessiveFailures = 0;
            }
        } catch (InterruptedException e) {
            LOG.info("LazyWriter was interrupted, exiting");
            break;
        } catch (Exception e) {
            LOG.warn("Ignoring exception in LazyWriter:", e);
        }
    }
}
}

```

之后，进入 saveNextReplica 方法的处理：

```

private boolean saveNextReplica() {
    RamDiskReplica block = null;
    FsVolumeReference targetReference;
    FsVolumeImpl targetVolume;
    ReplicaInfo replicaInfo;
    boolean succeeded = false;

    try {
        // 从队列中取出新的待持久化的块
        block = ramDiskReplicaTracker.dequeueNextReplicaToPersist();
        if (block != null) {
            synchronized (FsDatasetImpl.this) {
                ...
            }
        }
    }
}

```



```

// 提交到异步服务中去
asyncLazyPersistService.submitLazyPersistTask(
    block.getBlockPoolId(), block.getBlockId(),
    replicaInfo.getGenerationStamp(), block.getCreationTime(),
    replicaInfo.getMetaFile(), replicaInfo.getBlockFile(),
    targetReference);
}
}
}
succeeded = true;
} catch(IOException ioe) {
    LOG.warn("Exception saving replica " + block, ioe);
} finally {
    if (!succeeded && block != null) {
        LOG.warn("Failed to save replica " + block + ". re-enqueueing it.");
        // 进行副本块提交失败处理, 此副本块将会再次提交到待持久化队列中
        onFailLazyPersist(block.getBlockPoolId(), block.getBlockId());
    }
}
return succeeded;
}

```

LazyWriter 线程服务的流程图可以归纳为图 1-6。

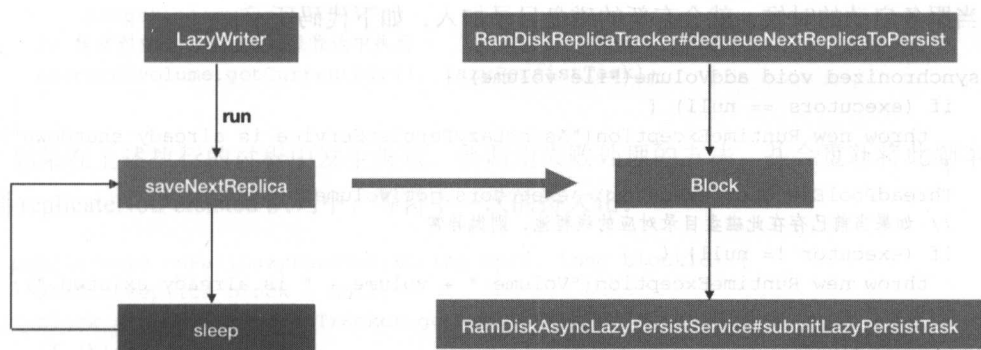


图 1-6 LazyWriter 服务流程图

我们结合 LazyWriter 和 RamDiskReplicaTracker 跟踪服务，就可以得到下面一个完整的流程（暂且不考虑 RamDiskAsyncLazyPersistService 的内部执行），如图 1-7 所示。

（3）RamDiskAsyncLazyPersistService

最后一部分异步服务的内容相对就比较简单了，主要围绕着 Volume 磁盘和 Executor 线程池这两部分的内容，秉持着下面一个原则：

一个磁盘服务对应一个线程池，并且一个线程池的最大线程数也只有 1 个。

线程池列表定义如下：

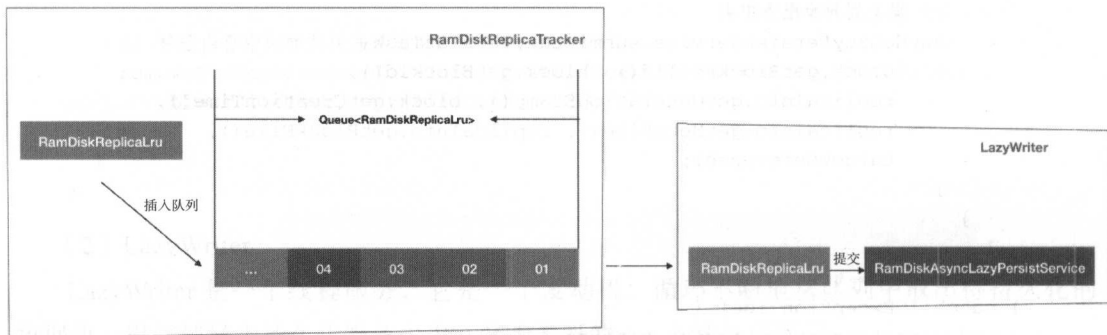


图 1-7 异步持久化流程图

```

class RamDiskAsyncLazyPersistService {
...
    private Map<File, ThreadPoolExecutor> executors
        = new HashMap<File, ThreadPoolExecutor>();
...

```

这里的 File 代表一个磁盘上的目录，个人认为这里完全可以用 String 字符串替代。既可以减少存储空间，又直观明了。从这里可以看出磁盘服务与线程池一对一的关系了。

当服务启动的时候，就会有新的磁盘目录加入，如下代码所示：

```

synchronized void addVolume(File volume) {
    if (executors == null) {
        throw new RuntimeException("AsyncLazyPersistService is already shutdown");
    }
    ThreadPoolExecutor executor = executors.get(volume);
    // 如果当前已存在此磁盘目录对应的线程池，则抛异常
    if (executor != null) {
        throw new RuntimeException("Volume " + volume + " is already existed.");
    }
    // 否则进行添加
    addExecutorForVolume(volume);
}

```

之后，进入 addExecutorForVolume 方法：

```

private void addExecutorForVolume(final File volume) {
...
    // 新建线程池，最大线程执行数为 1
    ThreadPoolExecutor executor = new ThreadPoolExecutor(
        CORE_THREADS_PER_VOLUME, MAXIMUM_THREADS_PER_VOLUME,
        THREADS_KEEP_ALIVE_SECONDS, TimeUnit.SECONDS,
        new LinkedBlockingQueue<Runnable>(), threadFactory);

    executor.allowCoreThreadTimeOut(true);
    // 加入到 executors 中，以 volume 作为 key

```

```

    executors.put(volume, executor);
}

```

还有一个需要注意的地方是提交执行方法 `submitLazyPersistTask`，如下所示：

```

void submitLazyPersistTask(String bpId, long blockId,
    long genStamp, long creationTime,
    File metaFile, File blockFile,
    FsVolumeReference target) throws IOException {
    if (LOG.isDebugEnabled()) {
        LOG.debug("LazyWriter schedule async task to persist RamDisk block pool id: "
            + bpId + " block id: " + blockId);
    }
    // 获取需要持久化的目标磁盘实例
    FsVolumeImpl volume = (FsVolumeImpl)target.getVolume();
    File lazyPersistDir = volume.getLazyPersistDir(bpId);
    if (!lazyPersistDir.exists() && !lazyPersistDir.mkdirs()) {
        FsDatasetImpl.LOG.warn("LazyWriter failed to create " + lazyPersistDir);
        throw new IOException("LazyWriter fail to find or create lazy persist dir: "
            + lazyPersistDir.toString());
    }
    // 新建此服务 Task
    ReplicaLazyPersistTask lazyPersistTask = new ReplicaLazyPersistTask(
        bpId, blockId, genStamp, creationTime, blockFile, metaFile,
        target, lazyPersistDir);
    // 提交到对应 volume 的线程池中执行
    execute(volume.getCurrentDir(), lazyPersistTask);
}

```

如果在上述执行的过程中发生失败，会调用失败处理的方法，并会重新将此副本块插入到 `replicateNotPersisted` 队列中，等待下一次的持久化：

```

public void onFailLazyPersist(String bpId, long blockId) {
    RamDiskReplica block = null;
    block = ramDiskReplicaTracker.getReplica(bpId, blockId);
    if (block != null) {
        LOG.warn("Failed to save replica " + block + ". re-enqueueing it.");
        // 重新插入队列操作
        ramDiskReplicaTracker.reenqueueReplicaNotPersisted(block);
    }
}

```

其他如 `removeVolume` 等方法实现比较简单，这里不做过多介绍。图 1-8 是 `RamDisk-AsyncLazyPersistService` 总的结构图。

以上 3 部分描述了 LAZT_PERSIST 下的队列式内存数据块持久化服务、异步持久化服务的内部运行逻辑和 LRU 预留内存空间算法策略。

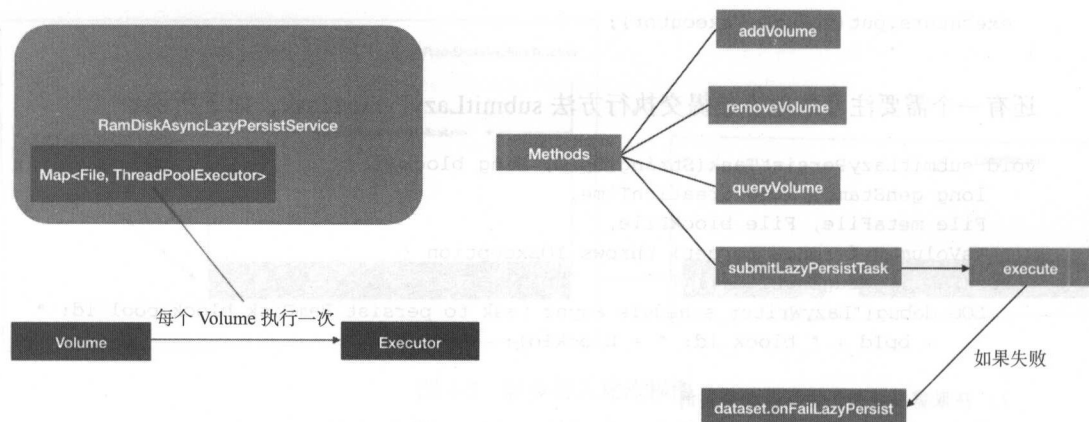


图 1-8 RamDiskAsyncLazyPersistService 相关结构图

1.1.4 LAZY_PERSIST 内存存储的使用

介绍完原理部分之后，下面介绍具体的配置使用。

第一步，要使用 LAZY_PERSIST 内存存储策略，需要有对应的存储介质，内存存储介质对应的类型是 RAM_DISK。

在使用 RAM_DISK 之前，需要完成虚拟内存盘的配置工作，这里以 tmpfs 文件系统为例进行介绍。在默认情况下，tmpfs 是被挂载到 /dev/shm，并且大小是 32GB。也就是说，在此目录下的数据实质上是存在于内存中的。但是有的时候，我们可能会想挂载到自己想要挂载的目录下，而且我们也想对内存的使用大小进行有效的控制，可以使用下面的命令进行这 2 方面的设置：

```
sudo mount -t tmpfs -o size=16g tmpfs /mnt/dn-tmpfs/
```

以上操作的意思是将 tmpfs 挂载到目录 /mnt/dn-tmpfs，并且限制内存使用大小为 16GB。最后，建议在 /etc/fstab 文件中将这层挂载关系写入，这可以让机器在重启之后自动创建好挂载关系。

首先需要将机器中已经完成好的虚拟内存盘配置到 dfs.datanode.data.dir 中，其次还要带上 RAM_DISK 标签，以此表明此目录对应的存储介质为 RAM_DISK，配置样例如下：

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>grid/0,/grid/1,/grid/2,[RAM_DISK]/mnt/dn-tmpfs</value>
</property>
```

注意，这个标签是必须要打上的，否则 HDFS 默认的都是 DISK。

第二步就是设置具体的文件策略类型。



注意

- 确保 HDFS 异构存储策略没有被关闭，默认是开启的，配置项是 `dfs.storage.policy.enabled`。
- 确认 `dfs.datanode.max.locked.memory` 是否设置了足够大的内存值，是否已是 `DataNode` 能承受的最大内存大小。内存值过小会导致内存中的总的可存储的数据块变少，但如果超过 `DataNode` 能承受的最大内存大小的话，部分内存块会被直接移出。

FsDatasetAsyncDiskService

在 `FsDatasetImpl` 类中，还有一个与内存存储异步持久化相类似的服务 `FsDatasetAsyncDiskService`。在类的实现逻辑上，该服务与 `RamDiskAsyncLazyPersistService` 有许多相似之处，同样包含许多执行线程池，并且每个线程池对应一个存储目录。不过在具体的执行内容上，它主要做两类异步任务：

- 文件目录的异步删除。
- 异步执行 `SyncFileRange` 请求操作。`SyncFileRange` 的全称是 `sync a file segment with disk`，它的作用在于数据做多次更新后，对其进行一次性的写出，提高 IO 的效率。

`FsDatasetAsyncDiskService` 类位于包 `org.apache.hadoop.hdfs.server.datanode.fsdataset.impl` 下，感兴趣的读者可以自行研究。

在 HDFS 异构存储方式中，除了内存存储之外，其实还有另外一类存储方式也尤为重要，就是 HDFS 的 Archival Storage。Archival Storage 指的是一种高密度的存储方式，以此解决集群数据规模增长带来的存储空间不足的问题。通常用于 Archival Storage 的节点不需要很好的计算性能，一般用于冷数据的存储。HDFS 的 Archival Storage 的具体设计与实现，可以参阅相关 JIRA，HDFS-6584(Support Archival Storage)。

1.2 HDFS 异构存储

Hadoop 在 2.6.0 版本中引入了一个新特性：异构存储。异构存储关键在于“异构”两个字。异构存储可以根据各个存储介质读写特性的不同发挥各自的优势。一个很适用的场景就是上节提到的冷热数据的存储。针对冷数据，采用容量大的、读写性能不高的存储介质存储，比如最普通的磁盘。而对于热数据而言，可以采用 SSD 的方式进行存储，这样就能保证高效的读性能，在速率上甚至能做到十倍或百倍于普通磁盘的读写速度。换句话说，

HDFS 异构存储特性的出现使得我们不需要搭建 2 套独立的集群来存放冷热 2 类数据，在一套集群内就能完成。所以这个功能还是有非常大的实用价值的。本节就带领大家全面了解 HDFS 的异构存储，包括异构存储的类型、存储策略、HDFS 如何做到智能化的异构存储等。

1.2.1 异构存储类型

以下是在 HDFS 中声明的 Storage Type:

❑ RAM_DISK

❑ SSD

❑ DISK

❑ ARCHIVE

HDFS 中定义了这 4 种异构存储类型，SSD、DISK 一看就知道是什么意思，这里看一下其余的两个。RAM_DISK 其实就是内存，而 ARCHIVE 并没有特指哪种存储介质，主要指的是高密度存储介质，用于解决数据扩容的问题。这 4 种类型定义在 StorageType 类中，如下所示：

```
public enum StorageType {
    // 根据存储的速度，从快到慢
    RAM_DISK(true),
    SSD(false),
    DISK(false),
    ARCHIVE(false);
    ...
}
```

其中 true 或者 false 代表此类存储类型是否为 transient 特性。transient 的意思是转瞬即逝的，并非持久化的。在上述 4 类介质中，只有内存存储才是 transient 特性的。在 HDFS 中，如果没有主动声明数据目录存储类型，默认都是 DISK 类型。这 4 类存储介质之间一个很大的区别在于读写速度，从上到下依次减慢。所以将热数据存在内存中或是 SSD 中会是不错的选择，而将冷数据存放于 DISK 和 ARCHIVE 类型的介质中会更好。在 HDFS 中，StorageType 的设定非常重要。那么如何让 HDFS 知道集群中的数据存储目录分别是哪种类型的存储介质呢？这就需要在配置属性时主动声明，HDFS 并没有自动检测识别的功能。配置属性 dfs.datanode.data.dir 可以对本地对存储目录进行设置，同时带上一个存储类型标签，声明此目录用的是哪种类型的存储介质，例子如下：

```
[SSD]file:///grid/dn/ssd0
```

如果目录前没有带上 [SSD]/[DISK]/[ARCHIVE]/[RAM_DISK] 这 4 种类型中的任何一种，则默认是 DISK 类型。图 1-9 是存储介质结构图。

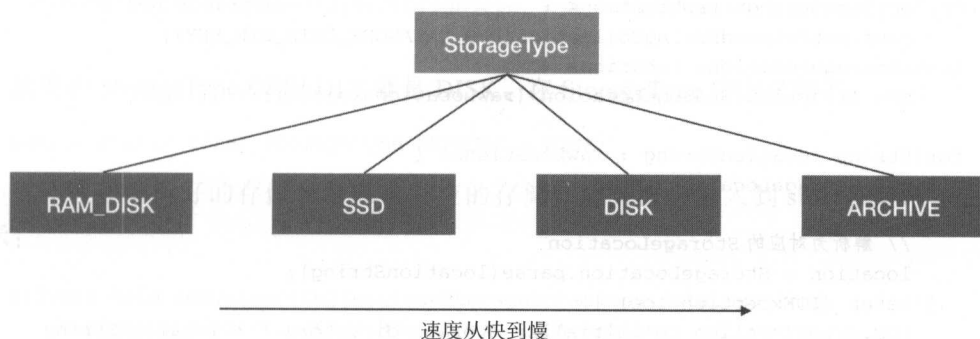


图 1-9 HDFS 存储介质类型

1.2.2 异构存储原理

了解完异构存储类型后，我们有必要了解一下 HDFS 异构存储的实现原理。本节会结合部分 HDFS 源码进行阐述。HDFS 异构存储可总结为以下三点：

- ❑ DataNode 通过心跳汇报自身数据存储目录的 StorageType 给 NameNode。
- ❑ 随后 NameNode 进行汇总并更新集群内各个节点的存储类型情况。
- ❑ 待复制文件根据自身设定的存储策略信息向 NameNode 请求拥有此类型存储介质的 DataNode 作为候选节点。

从以上 3 点来看，HDFS 异构存储原理并不复杂。下面结合部分源码，来一步步跟踪内部的过程细节。

1. DataNode 存储目录汇报

首先是数据存储目录的解析与心跳汇报过程。在 FsDatasetImpl 的构造函数中对 dataDir 进行存储目录的解析，生成了 StorageType 的 List 列表：

```
// FsDatasetImpl 初始构造函数
FsDatasetImpl(DataNode datanode, DataStorage storage, Configuration conf) throws IOException {
    ...
    String[] dataDirs = conf.getTrimmedStrings(DFSConfigKeys.DFS_DATANODE_DATA_DIR_KEY);
    Collection<StorageLocation> dataLocations = DataNode.getStorageLocations(conf);
    List<VolumeFailureInfo> volumeFailureInfos = getInitialVolumeFailureInfos(
        dataLocations, storage);
    ...
}
```

真正调用的是 DataNode 的 getStorageLocations 方法：

```
public static List<StorageLocation> getStorageLocations(Configuration conf) {
    // 获取 dfs.datanode.data.dir 配置中的多个目录地址字符串
}
```

```

Collection<String> rawLocations =
    conf.getTrimmedStringCollection(DFS_DATANODE_DATA_DIR_KEY);
List<StorageLocation> locations =
    new ArrayList<StorageLocation>(rawLocations.size());

for(String locationString : rawLocations) {
    final StorageLocation location;
    try {
        // 解析为对应的 StorageLocation
        location = StorageLocation.parse(locationString);
    } catch (IOException ioe) {
        LOG.error("Failed to initialize storage directory " + locationString
            + ". Exception details: " + ioe);
        // 此处忽略异常
        continue;
    } catch (SecurityException se) {
        LOG.error("Failed to initialize storage directory " + locationString
            + ". Exception details: " + se);
        // 此处忽略异常
        continue;
    }
    // 将解析好的 StorageLocation 加入到列表中
    locations.add(location);
}

return locations;
}

```

当然我们最关心如何解析配置并最终得到对应存储类型的过程，即下面这行操作所执行的内容：

```
location = StorageLocation.parse(locationString);
```

StorageLocation 的解析方法如下：

```

public static StorageLocation parse(String rawLocation)
    throws IOException, SecurityException {
    // 采用正则匹配的方式进行解析
    Matcher matcher = regex.matcher(rawLocation);
    StorageType storageType = StorageType.DEFAULT;
    String location = rawLocation;

    if (matcher.matches()) {
        String classString = matcher.group(1);
        location = matcher.group(2);
        if (!classString.isEmpty()) {
            storageType =
                StorageType.valueOf(StringUtils.toUpperCase(classString));
        }
    }
}

```

```

    return new StorageLocation(storageType, new Path(location).toUri());
}

```

这里的 `StorageType.DEFAULT` 就是 `DISK`，在 `StorageType` 中定义如下：

```
public static final StorageType DEFAULT = DISK;
```

后续这些解析好的存储目录以及对应的存储介质类型会加入到 `storageMap` 中，如下所示：

```

private void addVolume(Collection<StorageLocation> dataLocations,
    Storage.StorageDirectory sd) throws IOException {
    final File dir = sd.getCurrentDir();
    final StorageType storageType =
        getStorageTypeFromLocations(dataLocations, sd.getRoot());

    ...

    synchronized (this) {
        volumeMap.addAll(tempVolumeMap);
        storageMap.put(sd.getStorageUuid(),
            new DatanodeStorage(sd.getStorageUuid(),
                DatanodeStorage.State.NORMAL,
                storageType));
        ...
    }
}

```

`storageMap` 存储了目录到类型的映射关系，可以说是非常细粒度的。更重要的是，这些信息会被 `DataNode` 组织成 `StorageReport` 通过心跳的形式上报给 `NameNode`。于是就来到了第一阶段的下半过程：

```

public StorageReport[] getStorageReports(String bpid)
    throws IOException {
    List<StorageReport> reports;
    synchronized (statsLock) {
        List<FsVolumeImpl> curVolumes = getVolumes();
        reports = new ArrayList<>(curVolumes.size());
        for (FsVolumeImpl volume : curVolumes) {
            try (FsVolumeReference ref = volume.obtainReference()) {
                // 获取磁盘存储信息，并生成磁盘报告实例
                StorageReport sr = new StorageReport(volume.toDatanodeStorage(),
                    false,
                    volume.getCapacity(),
                    volume.getDfsUsed(),
                    volume.getAvailable(),
                    volume.getBlockPoolUsed(bpid));
                // 将报告实例加入到报告列表中
                reports.add(sr);
            } catch (ClosedChannelException e) {
                continue;
            }
        }
    }
}

```



```

    }
}
// 返回报告列表
return reports.toArray(new StorageReport[reports.size()]);
}

```

以上是 StorageReport 的组织过程，它最终被 BPServiceActor 的 sendHeartBeat 调用，发送给 NameNode，如下所示：

```

HeartbeatResponse sendHeartBeat() throws IOException {
    // 获取存储类型情况报告信息
    StorageReport[] reports =
        dn.getFSDataset().getStorageReports(bpos.getBlockPoolId());
    if (LOG.isDebugEnabled()) {
        LOG.debug("Sending heartbeat with " + reports.length +
            " storage reports from service actor: " + this);
    }
    // 获取坏磁盘数据信息
    VolumeFailureSummary volumeFailureSummary = dn.getFSDataset()
        .getVolumeFailureSummary();
    int numFailedVolumes = volumeFailureSummary != null ?
        volumeFailureSummary.getFailedStorageLocations().length : 0;
    // 还有 DataNode 自身的存储容量信息，最后发送给 NameNode
    return bpNamenode.sendHeartbeat(bpRegistration,
        reports,
        dn.getFSDataset().getCacheCapacity(),
        dn.getFSDataset().getCacheUsed(),
        dn.getXmitsInProgress(),
        dn.getXceiverCount(),
        numFailedVolumes,
        volumeFailureSummary);
}

```

2. 存储心跳信息的更新处理

现在来到了第二阶段的心跳处理过程。心跳处理在 DatanodeManager 的 handleHeartbeat 中进行：

```

// 心跳处理方法
public DatanodeCommand[] handleHeartbeat(DatanodeRegistration nodeReg,
    StorageReport[] reports, final String blockPoolId,
    long cacheCapacity, long cacheUsed, int xceiverCount,
    int maxTransfers, int failedVolumes,
    VolumeFailureSummary volumeFailureSummary) throws IOException {
    synchronized (heartbeatManager) {
        synchronized (datanodeMap) {
            DatanodeDescriptor nodeinfo = null;
            ...

```

```

heartbeatManager.updateHeartbeat(nodeinfo, reports,
                                cacheCapacity, cacheUsed,
                                xceiverCount, failedVolumes,
                                volumeFailureSummary);
...

```

最终在 heartbeatManager 中会调用到 DatanodeDescription 对象的 updateHeartbeatState 方法，该方法会更新 Storage 的信息，如下所示：

```

// 处理心跳中统计值相关的操作
public void updateHeartbeatState(StorageReport[] reports, long cacheCapacity,
                                long cacheUsed, int xceiverCount, int volFailures,
                                VolumeFailureSummary volumeFailureSummary) {
    ...
    for (StorageReport report : reports) {
        DatanodeStorageInfo storage = updateStorage(report.getStorage());
        if (checkFailedStorages) {
            failedStorageInfos.remove(storage);
        }

        storage.receivedHeartbeat(report);
        // 进行统计计数的更新统计
        totalCapacity += report.getCapacity();
        totalRemaining += report.getRemaining();
        totalBlockPoolUsed += report.getBlockPoolUsed();
        totalDfsUsed += report.getDfsUsed();
    }
    rollBlocksScheduled(getLastUpdateMonotonic());
    ...
}

```

3. 目标存储介质类型节点的请求

各个 DataNode 心跳信息都更新完毕之后，有目标存储介质需求的待复制文件块就会向 NameNode 请求 DataNode，这部分处理在 FSNamesystem 的 getAdditionalDatanode 中进行：

```

// 获取剩余块方法
LocatedBlock getAdditionalDatanode(String src, long fileId,
    final ExtendedBlock blk, final DatanodeInfo[] existings,
    final String[] storageIDs,
    final Set<Node> excludes,
    final int numAdditionalNodes, final String clientName
) throws IOException {
    ...
    final INodeFile file = checkLease(src, clientName, inode, fileId);
    clientMachine = file.getFileUnderConstructionFeature().getClientMachine();
    clientnode = blockManager.getDatanodeManager().getDatanodeByHost(clientMachine);
    preferredblocksize = file.getPreferredBlockSize();
    // 获取待复制文件的存储策略 Id，对应的就是存储策略信息类型
    storagePolicyID = file.getStoragePolicyID();
}

```

```

// 寻找存储目录信息
final DatanodeManager dm = blockManager.getDatanodeManager();
// 获取已存在节点的存储目录列表信息
chosen = Arrays.asList(dm.getDatanodeStorageInfos(existings, storageIDs));
} finally {
    readUnlock();
}

...
// 选择满足需求的节点
final DatanodeStorageInfo[] targets = blockManager.chooseTarget4AdditionalDatanode(
    src, numAdditionalNodes, clientnode, chosen,
    excludes, preferredblocksize, storagePolicyID);
final LocatedBlock lb = new LocatedBlock(blk, targets);
blockManager.setBlockToken(lb, AccessMode.COPY);
return lb;
}

```

目标存储节点信息就被设置到了具体块的信息中。这里的 target 类型为 DatanodeStorageInfo，代表的是 DataNode 中的一个 dataDir 存储目录。上述代码中具体 blockManager 如何根据给定的候选 DatanodeStorageInfo 存储目录和存储策略来选择出目标节点，就是下一节将要重点阐述的存储介质选择策略。本节最后给出 HDFS 的异构存储过程调用的简单流程图，如图 1-10 所示。

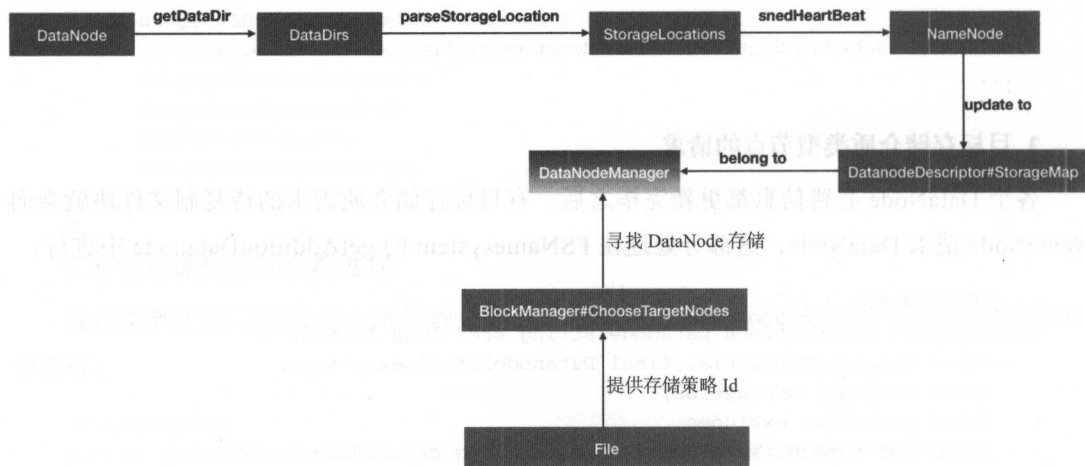


图 1-10 异构存储过程图

1.2.3 块存储类型选择策略

在现有的 HDFS 中，我们可以对块的网络拓扑位置进行策略的选择，同样，对于数据的存储介质，HDFS 也有对应的若干种策略。对于一个完整的存储类型选择策略，有如下的

基本信息定义:

```
// 块存储类型选择策略对象
@interfaceAudience.Private
public class BlockStoragePolicy {
    public static final Logger LOG = LoggerFactory.getLogger(BlockStoragePolicy
        .class);

    // 策略唯一标识 Id
    private final byte id;
    // 策略名称
    private final String name;

    // 对于一个新块, 存储副本块的可选存储类型信息组
    private final StorageType[] storageTypes;
    // 对于第一个创建块, fallback 情况时的可选存储类型
    private final StorageType[] creationFallbacks;
    // 对于块的其余副本, fallback 情况时的可选存储类型
    private final StorageType[] replicationFallbacks;

    // 当创建文件的时候, 是否继承祖先目录信息的策略, 主要用于主动设置策略的时候
    private boolean copyOnCreateFile;
    ...
}
```

这里出现了 fallback 的情况, 什么叫做 fallback 的情况呢? 即当前存储类型不可用的时候, 退一级选择使用的存储类型。

相应的逻辑代码如下:

```
public List<StorageType> chooseStorageTypes(final short replication,
    final Iterable<StorageType> chosen,
    final EnumSet<StorageType> unavailables,
    final boolean isNewBlock) {
    ...
    for(int i = storageTypes.size() - 1; i >= 0; i--) {
        // 获取当前需要的存储类型
        final StorageType t = storageTypes.get(i);
        // 如果当前的存储类型是在不可用的存储类型列表中, 选择 fallback 的情况
        if (unavailables.contains(t)) {
            // 根据是否为新块还是普通的副本块, 选择相应的 fallback 的 StorageType
            final StorageType fallback = isNewBlock?
                getCreationFallback(unavailables)
                : getReplicationFallback(unavailables);
            if (fallback == null) {
                removed.add(storageTypes.remove(i));
            } else {
                storageTypes.set(i, fallback);
            }
        }
    }
    ...
}
```

在 `getFallback` 方法中会选取第一个满足条件的 `fallback` 的 `StorageType`:

```
private static StorageType getFallback(EnumSet<StorageType> unavailables,
    StorageType[] fallbacks) {
    for(StorageType fb : fallbacks) {
        // 如果找到满足条件的 StorageType, 立即返回
        if (!unavailables.contains(fb)) {
            return fb;
        }
    }
    return null;
}
```

当然这些都只是单一的存储类型选择策略。HDFS 在使用的时候也不是新建一个 `StoragePolicy` 对象直接调用, 而是从 `BlockStoragePolicySuite` 策略集合中获取策略。

1.2.4 块存储策略集合

块存储策略集合是 `BlockStoragePolicySuite`。在此类内部定义了 6 种策略, 不仅仅分为冷热数据两种类型, 其详细策略描述可见源代码中的解释。类策略名称如下:

❑ HOT

❑ COLD

❑ WARM

❑ ALL_SSD

❑ ONE_SSD

❑ LAZY_PERSIST

在这 6 种策略中, 前三种策略和后三种策略可以看作是两大类。前三种策略是根据冷热数据的角度来区分的, 后三种策略是根据存放盘的性质来区分的。策略倒是划分出来了, 但是这些不同的策略之间的主要区别在哪里呢, 答案就是候选存储类型组。

在创建 `BlockStoragePolicySuite` 的时候, 对这些策略都进行了构造, 如下所示:

```
// 块存储策略集的构造初始化
public static BlockStoragePolicySuite createDefaultSuite() {
    final BlockStoragePolicy[] policies =
        new BlockStoragePolicy[1 << ID_BIT_LENGTH];
    final byte lazyPersistId = HdfsConstants.MEMORY_STORAGE_POLICY_ID;
    // LAZY_PERSIST 策略构造
    policies[lazyPersistId] = new BlockStoragePolicy(lazyPersistId,
        HdfsConstants.MEMORY_STORAGE_POLICY_NAME,
        new StorageType[]{StorageType.RAM_DISK, StorageType.DISK},
        new StorageType[]{StorageType.DISK},
        new StorageType[]{StorageType.DISK},
        true);    // Cannot be changed on regular files, but inherited.
```

```

final byte allssdId = HdfsConstants.ALLSSD_STORAGE_POLICY_ID;
// ALL_SSD 策略构造
policies[allssdId] = new BlockStoragePolicy(allssdId,
    HdfsConstants.ALLSSD_STORAGE_POLICY_NAME,
    new StorageType[]{StorageType.SSD},
    new StorageType[]{StorageType.DISK},
    new StorageType[]{StorageType.DISK});
final byte onessdId = HdfsConstants.ONESSD_STORAGE_POLICY_ID;
// ONE_SSD 策略构造
policies[onessdId] = new BlockStoragePolicy(onessdId,
    HdfsConstants.ONESSD_STORAGE_POLICY_NAME,
    new StorageType[]{StorageType.SSD, StorageType.DISK},
    new StorageType[]{StorageType.SSD, StorageType.DISK},
    new StorageType[]{StorageType.SSD, StorageType.DISK});
final byte hotId = HdfsConstants.HOT_STORAGE_POLICY_ID;
// HOT 策略构造
policies[hotId] = new BlockStoragePolicy(hotId,
    HdfsConstants.HOT_STORAGE_POLICY_NAME,
    new StorageType[]{StorageType.DISK}, StorageType.EMPTY_ARRAY,
    new StorageType[]{StorageType.ARCHIVE});
final byte warmId = HdfsConstants.WARM_STORAGE_POLICY_ID;
// WARM 策略构造
policies[warmId] = new BlockStoragePolicy(warmId,
    HdfsConstants.WARM_STORAGE_POLICY_NAME,
    new StorageType[]{StorageType.DISK, StorageType.ARCHIVE},
    new StorageType[]{StorageType.DISK, StorageType.ARCHIVE},
    new StorageType[]{StorageType.DISK, StorageType.ARCHIVE});
final byte coldId = HdfsConstants.COLD_STORAGE_POLICY_ID;
// COLD 策略构造
policies[coldId] = new BlockStoragePolicy(coldId,
    HdfsConstants.COLD_STORAGE_POLICY_NAME,
    new StorageType[]{StorageType.ARCHIVE}, StorageType.EMPTY_ARRAY,
    StorageType.EMPTY_ARRAY);
return new BlockStoragePolicySuite(hotId, policies);
}

```

在这些策略对象的参数中，第三个参数起决定性作用，因为第三个参数会被返回给副本块作为候选存储类型。在 `storageTypes` 参数中，有时可能只有 1 个类型声明，例如 ALL_SSD 策略只有如下 `StorageType`：

```
new StorageType[]{StorageType.SSD}
```

而 ONE_SSD 却有两个：

```
new StorageType[]{StorageType.SSD, StorageType.DISK}
```

这里面其实是有原因的。因为块有多副本机制，每个策略要为所有的副本都返回相应的 `StorageType`，如果副本数超过候选的 `StorageType` 数组时应怎么处理，答案在下面这个方法中：

```

public List<StorageType> chooseStorageTypes(final short replication) {
    final List<StorageType> types = new LinkedList<StorageType>();
    int i = 0, j = 0;

    // 从前往后依次匹配存储类型与对应的副本下标相匹配, 同时要过滤掉
    // transient 属性的存储类型
    for (; i < replication && j < storageTypes.length; ++j) {
        if (!storageTypes[j].isTransient()) {
            types.add(storageTypes[j]);
            ++i;
        }
    }

    // 获取最后一个存储类型, 统一作为多余副本的存储类型
    final StorageType last = storageTypes[storageTypes.length - 1];
    if (!last.isTransient()) {
        for (; i < replication; i++) {
            types.add(last);
        }
    }
    return types;
}

```

这样的话, ONE_SSD 就必然只有第一个块的副本块是此类型的, 其余副本则是 DISK 类型存储, 而 ALL_SSD 则将会全部是 SSD 的存储。图 1-11 给出存储策略集合的结构图。

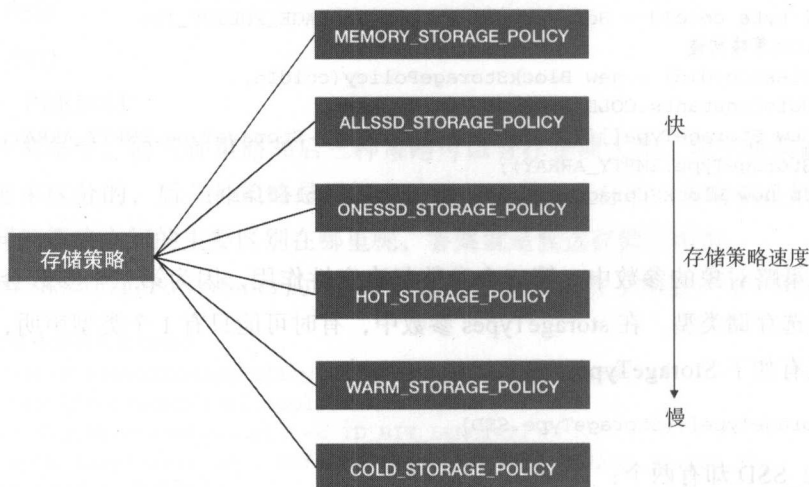


图 1-11 存储策略集合

上述策略中有一个策略在前面提到过, 就是 LAZY_PERSIST。此策略在执行的时候会将数据写到内存中, 然后再持久化。大家可以试试此策略, 看看性能到底如何。

1.2.5 块存储策略的调用

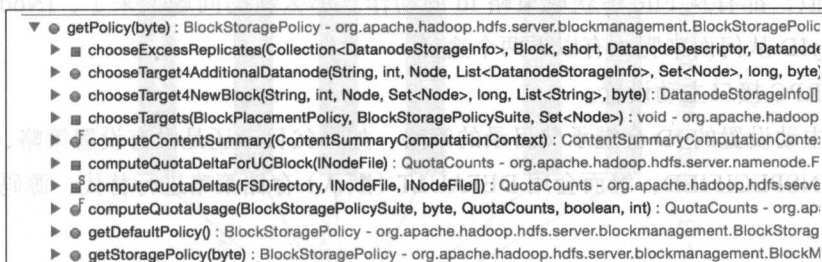
分析完块存储策略的种类之后，我们看看 HDFS 在哪些地方设置了这些策略。

首先，我们要知道 HDFS 的默认策略是哪种，默认策略如下：

```
@VisibleForTesting
public static BlockStoragePolicySuite createDefaultSuite() {
    ...
    return new BlockStoragePolicySuite(hotId, policies);
}
...

public BlockStoragePolicySuite(byte defaultPolicyID,
    BlockStoragePolicy[] policies) {
    this.defaultPolicyID = defaultPolicyID;
    this.policies = policies;
}
```

可以看出，这就是 HOT 的策略。也就是说，在默认情况下，HDFS 把集群中的数据都看成是经常访问的数据。然后进一步查看 `getPolicy` 的方法调用，如图 1-12 所示。



```
▼ ● getPolicy(byte) : BlockStoragePolicy - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ■ chooseExcessReplicates(Collection<DatanodeStorageInfo>, Block, short, DatanodeDescriptor, DatanodeStorageInfo) : void - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ● chooseTarget4AdditionalDatanode(String, int, Node, List<DatanodeStorageInfo>, Set<Node>, long, byte) : DatanodeStorageInfo[] - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ● chooseTarget4NewBlock(String, int, Node, Set<Node>, long, List<String>, byte) : DatanodeStorageInfo[] - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ■ chooseTargets(BlockPlacementPolicy, BlockStoragePolicySuite, Set<Node>) : void - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► f computeContentSummary(ContentSummaryComputationContext) : ContentSummaryComputationContext - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ■ computeQuotaDeltaForUCBlock(INodeFile) : QuotaCounts - org.apache.hadoop.hdfs.server.namenode.FSQuota
    ► ■ computeQuotaDeltas(FSDirectory, INodeFile, INodeFile[]) : QuotaCounts - org.apache.hadoop.hdfs.server.namenode.FSQuota
    ► f computeQuotaUsage(BlockStoragePolicySuite, byte, QuotaCounts, boolean, int) : QuotaCounts - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ● getPolicy() : BlockStoragePolicy - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
    ► ● getStoragePolicy(byte) : BlockStoragePolicy - org.apache.hadoop.hdfs.server.blockmanagement.BlockStoragePolicy
```

图 1-12 `getPolicy` 方法调用

我们以方法 `chooseTarget4NewBlock` 为例，追踪一下上游的调用过程。

```
public DatanodeStorageInfo[] chooseTarget4NewBlock(final String src,
    final int numOfReplicas, final Node client,
    final Set<Node> excludedNodes,
    final long blockSize,
    final List<String> favoredNodes,
    final byte storagePolicyID) throws IOException {
    List<DatanodeDescriptor> favoredDatanodeDescriptors =
        getDatanodeDescriptors(favoredNodes);
    final BlockStoragePolicy storagePolicy = storagePolicySuite.getPolicy(
        (storagePolicyID));
    ...
}
```

在父方法中获取了 `StoragePolicyID` 策略 ID，往上追踪，来到了 `FSNamesystem` 的 `getNewBlockTargets` 方法：

```

DatanodeStorageInfo[] getNewBlockTargets(String src, long fileId,
    String clientName, ExtendedBlock previous, Set<Node> excludedNodes,
    List<String> favoredNodes, LocatedBlock[] onRetryBlock) throws IOException {
    ...
    replication = pendingFile.getFileReplication();
    storagePolicyID = pendingFile.getStoragePolicyID();
} finally {
    readUnlock();
}

if (clientNode == null) {
    clientNode = getClientNode(clientMachine);
}

// 为新分配的块选择目标节点
return getBlockManager().chooseTarget4NewBlock(
    src, replication, clientNode, excludedNodes, blockSize, favoredNodes,
    storagePolicyID);
}

```

于是我们看到 `StoragePolicyID` 是从 `INodeFile` 中获取而来的。这与上文中目标节点请求的过程类似，都有从 `File` 中获取策略 `Id` 的动作。那么新的问题又来了，`INodeFile` 中的 `StoragePolicyID` 从何而来呢，有以下两个途径：

- ❑ 通过 `RPC` 接口主动设置。
- ❑ 没有主动设置的 `ID` 会继承父目录的策略，如果父目录还是没有设置策略，则会设置 `ID_UNSPECIFIED`，继而会用 `DEFAULT`（默认）存储策略进行替代，源码如下：

```

...
public byte getStoragePolicyID() {
    byte id = getLocalStoragePolicyID();
    if (id == ID_UNSPECIFIED) {
        return this.getParent() != null ?
            this.getParent().getStoragePolicyID() : id;
    }
    return id;
}
...

```

综上，HDFS 异构存储总的过程调用见图 1-13。

1.2.6 HDFS 异构存储策略的不足之处

前面花了很多的篇幅介绍了 HDFS 各种异构存储策略的特点、优势以及过程调用，那么是否这套机制是完美无缺的呢？答案当然不是，下面场景就不适合使用此机制。

用户 A 在 HDFS 上创建自己的存储目录 `/user/A`，不设置任何的存储策略，也就是默认都存放在 `DISK` 类型的介质上。忽然有一天，他发现自己的数据已经不怎么使用了，想要

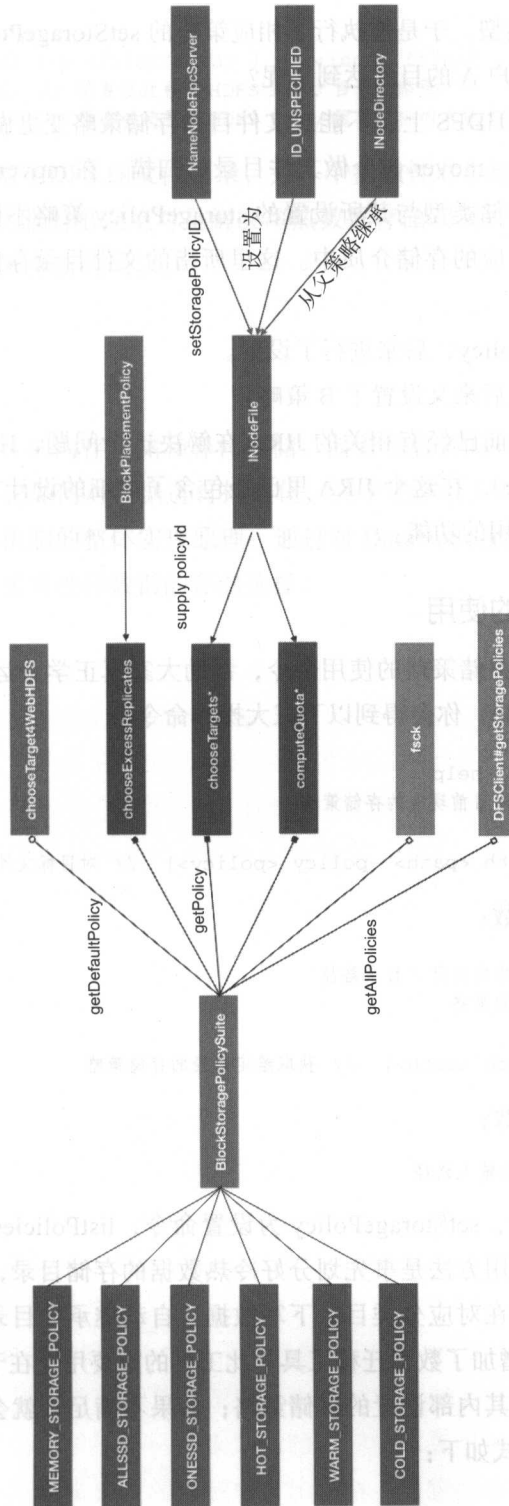


图 1-13 异构存储策略总的过程调用

设置其存储策略为 COLD 类型，于是他执行了相应策略的 `setStoragePolicy` 命令。那么这步命令操作完了是否意味着用户 A 的目的达到了呢？

问题就出在变更，目前 HDFS 上还不能对文件目录存储策略变更做出自动的数据迁移。这里需要用户额外执行 `hdfs -mover` 命令做文件目录的扫描。在 `mover` 命令扫描的过程中，如果发现文件目录的实际存储类型与其所设置的 `storagePolicy` 策略不同，将会进行数据块的迁移，将数据迁移到相对应的存储介质中。这里所指的文件目录存储策略变更有以下两类情况：

❑ 原先未设置 `StoragePolicy`，后来进行了设置。

❑ 原先设置了 A 策略，后来又设置了 B 策略。

针对这个问题，社区目前已经有相关的 JIRA 在解决这个问题，HDFS-10285 (`Storage Policy Satisfier in Namenode`)，在这个 JIRA 里已经包含了详细的设计文档，如果这个问题被解决了，将会是一个很实用的功能。

1.2.7 HDFS 存储策略的使用

本节最后介绍几个关于存储策略的使用命令，帮助大家真正学会运用这个强大的特性。输入 `hdfs storagepolicies -help`，你会得到以下三大操作命令：

```
$ hdfs storagepolicies -help
[-listPolicies] // 列出目前现有的存储策略

[-setStoragePolicy -path <path> -policy <policy>] // 对目标文件 / 目录设置存储策略
```

以下为此命令的必填参数：

```
<path>    需要设置存储策略的文件 / 目录路径
<policy>  对目标设置的存储策略
```

```
[-getStoragePolicy -path <path>] // 获取给定路径的存储策略
```

以下为此命令的必填参数：

```
<path>    需要获取存储策略的输入路径
```

在以上三大操作命令中，`setStoragePolicy` 为设置命令，`listPolicies` 和 `getStoragePolicy` 都是获取命令。最简单的使用方法是事先划分好冷热数据的存储目录，设置好对应的存储策略，后续使用相应的程序在对应分类目录下写数据，自动继承父目录的存储策略。在较新版的 Hadoop 发布版本中增加了数据迁移工具。此工具的重要用途在于它会扫描 HDFS 上的文件，判断文件是否满足其内部设置的存储策略；如果不满足，就会重新迁移数据到目标存储类型节点上。使用方式如下：

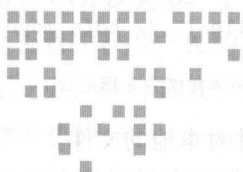
```
$ hdfs mover -help
Usage: hdfs mover [-p <files/dirs> | -f <local file>] // hdfs mover 数据迁移命令
-p <files/dirs> // 需要被迁移的 HDFS 文件 / 目录的路径
-f <local file> // 需要被迁移的 HDFS 文件 / 目录对应的本地文件系统路径
```

其中一个参数针对 HDFS 的文件目录，另一个参数针对本地的文件。

HDFS 异构存储功能的出现绝对是解决冷热数据存储问题的一把利器，希望通过本节内容的阐述能给大家带来全新的认识。

1.3 小结

本章介绍了 HDFS 的内存存储和异构存储，前者是后者的一种存储方式。HDFS 的内存存储方式的难点在于它如何对数据进行持久化，其中还涉及 LRU 算法。而 HDFS 异构存储的重点在于理解此套机制的整体实现原理，通过对 DataNode 上的数据目录以及 HDFS 上的目标路径打标签的方式来进行数据的存储选择。



HDFS 的数据管理与策略选择

本章我们要学习 HDFS 的数据管理，HDFS 有独特的数据管理方式和策略选择。本章首先介绍 HDFS 缓存方面的管理，包括缓存块与 DataNode 数据之间的交互，HDFS 整体的中心缓存机制的实现原理。第二介绍 HDFS 的快照管理，快照在很多应用的场景下可以用来进行数据的故障恢复，在 HDFS 中，它有相似的作用。第三介绍经典的三副本策略，及其在 HDFS 中如何做到数据的高可用性。第四介绍 HDFS 的 Sasl 认证与 Blocktoken 验证机制以及二者实现的差别。最后介绍 DataNode 内部的三大“数据保镖”服务：VolumeScanner、DirectoryScanner 以及 DiskChecker。

2.1 HDFS 缓存与缓存块

HDFS 的缓存与我们平常所说的缓存 (cache) 在作用上是一致的，主要是为了减少重复的数据请求过程。但是在具体实现上，我们平常所用的缓存可能只由一个简单的缓冲数组构成，而 HDFS 用的是缓存块 (cache block) 的概念。HDFS 的缓存块由普通的文件块转换而来，同样也可以转换回去。HDFS 缓存的出现可以大大提高用户读取文件的速度，因为它是缓存在 DataNode 内存中的，此过程无需进行读取磁盘的操作。在本节中，我们主要介绍缓存块的缓存原理、缓存块的周期状态以及缓存块不同状态之间的转换等内容。

笔者在学习 HDFS 缓存方面的原理过程中，感觉到其中的逻辑有点绕，直接分析不见得会起到很好的效果，所以这里采取疑问点的形式来做一个引导。在列出下面几个疑问点之前，需要了解一些相关名称的定义：在 HDFS 中缓存的对象是数据块，需要缓存的目标

数据块称为 CacheBlock，不需要缓存的数据块称为 UnCacheBlock。以下是本节所要答复的疑问点：

- 物理层面缓存块是怎样的？
- 缓存块的生命周期状态有哪几种？
- 哪些情况会触发缓存块、取消缓存块的操作？
- CacheBlock、UnCacheBlock 缓存块如何确定？
- 系统所持有的缓存块列表如何更新？
- 缓存块如何被使用？

下面将详细讲述这些内容。

2.1.1 HDFS 物理层面缓存块

物理层面缓存块这个词在 HDFS 源码中的解释如下：“利用 mmap、mlock 这样的系统调用将块数据锁入内存，以此达到在 DataNode 上缓存数据的效果。”大意为利用 mmap、mlock 系统调用将块锁入内存。没接触过底层操作系统知识的人可能不是很清楚 mmap、mlock 调用是怎么一回事，下面简单介绍一下。mmap 系统调用，它是一个内存映射调用。mmap 主要作用是将一个文件或者其他对象映射进内存。将文件或其他对象映射进内存存在代码中的体现如下所示：

```
// 加载并映射块到内存，然后检验其 checksum
public static MappableBlock load(long length,
    FileInputStream blockIn, FileInputStream metaIn,
    String blockFileName) throws IOException {
    MappableBlock mappableBlock = null;
    MappedByteBuffer mmap = null;
    FileChannel blockChannel = null;
    try {
        // 获取块数据的 FileChannel 对象
        blockChannel = blockIn.getChannel();
        if (blockChannel == null) {
            throw new IOException("Block InputStream has no FileChannel.");
        }
        // 这里开始进行内存的映射操作
        mmap = blockChannel.map(MapMode.READ_ONLY, 0, length);
        NativeIO.POSIX.getCacheManipulator().mlock(blockFileName, mmap, length);
        verifyChecksum(length, metaIn, blockChannel, blockFileName);
        mappableBlock = new MappableBlock(mmap, length);
    } finally {
        IOUtils.closeQuietly(blockChannel);
        if (mappableBlock == null) {
            if (mmap != null) {
                // 解除地址区域的对象映射
                NativeIO.POSIX.munmap(mmap);
            }
        }
    }
}
```

```

    }
    }
    return mappableBlock;
}

```

在上面的代码中，将 `blockChannel` 对象（本质对象是 `FileChannel`）映射到了内存中。当然这是最底层执行的操作了，在 HDFS 中的上层调用是如何的呢，这才是我们所要真正关心的，下面继续讲。

2.1.2 缓存块的生命周期状态

缓存块的生命周期不仅仅只有 `Cached`（已缓存）和 `UnCached`（未缓存）两种。在 `FSDatasetCache` 类中，有明确的定义：

```

private enum State {
    // 缓存块缓存中状态
    CACHING,

    // 缓存块取消状态
    CACHING_CANCELLED,

    // 缓存块已缓存状态
    CACHED,

    // 缓存块取消缓存状态
    UNCACHING;

    // 块是否已缓存判断方法
    public boolean shouldAdvertise() {
        return (this == CACHED);
    }
}

```

上面的 `Cache` 状态信息总共分为四类：

- ❑ `CACHING` 表示块正在被缓存。
- ❑ `CACHING_CANCELLED` 正在被缓存的块已处于被取消的状态。
- ❑ `CACHED` 表明数据块已被缓存。
- ❑ `UNCACHING` 表明缓存块正处于取消缓存的状态。

在 HDFS 的缓存过程中有这四类缓存状态，并可以切换。

这个状态信息保存在 `FsDatasetCache` 的 `Value` 内部类中：

```

// 缓存状态信息
private static final class Value {
    // 缓存状态对象

```



```

final State state;
// 缓存块对象
final MappableBlock mappableBlock;

Value(MappableBlock mappableBlock, State state) {
    this.mappableBlock = mappableBlock;
    this.state = state;
}
}

```

实际存储的 Value 对象与块 Id 对象构成了缓存映射图：

```

private final HashMap<ExtendedBlockId, Value> mappableBlockMap =
    new HashMap<ExtendedBlockId, Value>();

```

可能有人会问为什么这里不直接用 64 位的块 Id 直接当 key，而是用 ExtendedBlockId 对象，这是因为 BlockPool 的存在。现在的 HDFS 是支持多命名空间的，块 Id 只在同一个 BlockPool 下唯一。ExtendedBlockId、Value 键值对的存储与清除发生在 cacheBlock 和 uncacheBlock 方法内：

```

synchronized void cacheBlock(long blockId, String bpid,
    String blockFileName, long length, long genstamp,
    Executor volumeExecutor) {
    ...
    mappableBlockMap.put(key, new Value(null, State.CACHING));
    volumeExecutor.execute(
        new CachingTask(key, blockFileName, length, genstamp));
    LOG.debug("Initiating caching for Block with id {}, pool {}", blockId,
        bpid);
}

```

然后将块文件信息传入 CachingTask 中异步执行。同理 uncacheBlock 方法也放在了异步线程中执行：

```

synchronized void uncacheBlock(String bpid, long blockId) {
    ExtendedBlockId key = new ExtendedBlockId(blockId, bpid);
    // 获取当前缓存块对象
    Value prevValue = mappableBlockMap.get(key);
    boolean deferred = false;
    ...
    // 进行块对象缓存状态的判断
    switch (prevValue.state) {
    case CACHING:
        LOG.debug("Cancelling caching for block with id {}, pool {}. ", blockId,
            bpid);
        mappableBlockMap.put(key,
            new Value(prevValue.mappableBlock, State.CACHING_CANCELLED));
        break;
    }
}

```

```

case CACHED:
    // 如果是已缓存状态, 则将进行取消缓存操作
    mappableBlockMap.put(key,
        new Value(prevValue.mappableBlock, State.UNCACHING));
    if (deferred) {
        LOG.debug("{} is anchored, and can't be uncached now. Scheduling it " +
            "for uncaching in {} ",
            key, DurationFormatUtils.formatDurationHMS(revocationPollingMs));
        deferredUncachingExecutor.schedule(
            new UncachingTask(key, revocationMs,
                revocationPollingMs, TimeUnit.MILLISECONDS));
    } else {
        LOG.debug("{} has been scheduled for immediate uncaching.", key);
        uncachingExecutor.execute(new UncachingTask(key, 0));
    }
    break;
default:
    LOG.debug("Block with id {}, pool {} does not need to be uncached, "
        + "because it is in state {}.", blockId, bpid, prevValue.state);
    numBlocksFailedToUncache.incrementAndGet();
    break;
}
}
}

```

2.1.3 CacheBlock、UnCacheBlock 场景触发

什么下情况会触发缓存块的行为呢? 同样我们要将此情形分为两类: 一类是 CacheBlock, 另外一类是 UnCacheBlock。

1. CacheBlock 动作

在 Eclipse 编译器中通过 open call 的方式可以追踪出它的上层方法调用, 如图 2-1 所示。

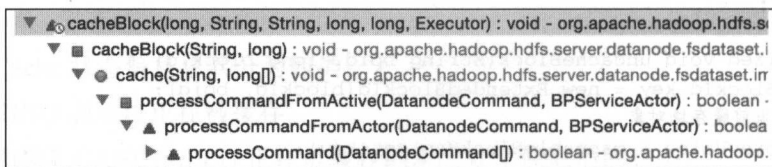


图 2-1 CacheBlock 调用

最下层的方法表明了此方法最终来自于 NameNode 心跳处理的方法, 进入此方法进行查阅:

```

private boolean processCommandFromActive(DatanodeCommand cmd,
    BPServiceActor actor) throws IOException {
    final BlockCommand bcmd =
        cmd instanceof BlockCommand? (BlockCommand)cmd: null;

```

```

final BlockIdCommand blockIdCmd =
cmd instanceof BlockIdCommand ? (BlockIdCommand)cmd : null;

switch(cmd.getAction()) {
...

// DataNode 获取到心跳返回的缓存命令
case DatanodeProtocol.DNA_CACHE:
LOG.info("DatanodeCommand action: DNA_CACHE for " +
        blockIdCmd.getBlockPoolId() + " of [ " +
        blockIdArrayToString(blockIdCmd.getBlockIds()) + " ]");
// 进行缓存块的动作, 目标缓存块列表也是从心跳回复中获得
dn.getFSDataset().cache(blockIdCmd.getBlockPoolId(), blockIdCmd.getBlockIds());
break;
...
default:
LOG.warn("Unknown DatanodeCommand action: " + cmd.getAction());
}
return true;
}

```

2. UnCacheBlock 动作

取消缓存块的动作是否也来自 NameNode 的回复命令呢? 答案其实不仅限于此。调用关系图如图 2-2 所示。

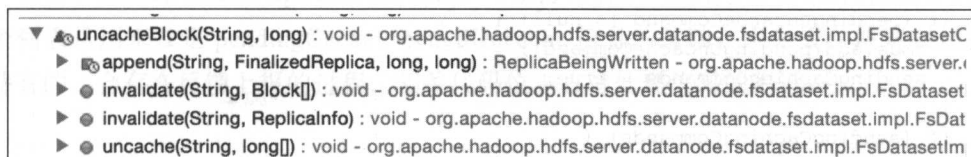


图 2-2 UnCache 调用

可以看到, 这里总共有三类调用情况:

- ❑ 当块执行 append 写操作的时候。
- ❑ 当把块处理为无效块的时候。
- ❑ 上层 NameNode 发送 uncache 回复命令的时候。

最后一种跟之前 CacheBlock 提到的情况一样, 都是由 NameNode 的回复命令所触发的。前面两类情况导致取消缓存动作的理由很好理解, 原因如下:

- ❑ 因为对块继续执行了写动作, 数据必然发生改变, 原有的缓存块需要重新更新。
- ❑ 当把块处理为无效块的时候, 接着会被 NameNode 从系统中清除, 缓存块自然而然就没有存在的必要了。

2.1.4 CacheBlock、UnCacheBlock 缓存块的确定

目标缓存块的确定问题本质上等同于 NameNode 回复命令中 CacheBlock、UnCacheBlock 的块 Id 的确定问题，就是如下代码中的 BlockPoolId 和 BlockIds：

```
dn.getFSDataset().cache(blockIdCmd.getBlockPoolId(), blockIdCmd.getBlockIds());
```

blockIdCmd 是 NameNode 心跳处理的回复命令，所以必然存在回复命令构造的过程。

这里进入到 DatanodeManager 的 handleHeartbeat 命令处理方法：

```
if (shouldSendCachingCommands &&
    ((nowMs - nodeinfo.getLastCachingDirectiveSentTimeMs()) >=
     timeBetweenResendingCachingDirectivesMs)) {
    // 构造需要缓存的块命令
    DatanodeCommand pendingCacheCommand =
        getCacheCommand(nodeinfo.getPendingCached(), nodeinfo,
            DatanodeProtocol.DNA_CACHE, blockPoolId);
    if (pendingCacheCommand != null) {
        cmds.add(pendingCacheCommand);
        sendingCachingCommands = true;
    }
    // 构造需要取消缓存的块命令
    DatanodeCommand pendingUncacheCommand =
        getCacheCommand(nodeinfo.getPendingUncached(), nodeinfo,
            DatanodeProtocol.DNA_UNCACHE, blockPoolId);
    if (pendingUncacheCommand != null) {
        cmds.add(pendingUncacheCommand);
        sendingCachingCommands = true;
    }
    if (sendingCachingCommands) {
        nodeinfo.setLastCachingDirectiveSentTimeMs(nowMs);
    }
}
```

从这里可以看出，CacheBlock 和 UnCacheBlock 来源于 nodeInfo 中的 pendingCache 和 pendingUncache 对象信息，实质上获取的变量如下：

```
// DataNode 上已缓存的缓存块列表
private final CachedBlocksList cached =
    new CachedBlocksList(this, CachedBlocksList.Type.CACHED);

// DataNode 上待取消缓存的缓存块列表
private final CachedBlocksList pendingUncached =
    new CachedBlocksList(this, CachedBlocksList.Type.PENDING_UNCACHED);
```

只要能找到 CachedBlockList 的直接操作方，就能明白缓存块、待取消缓存块是如何确定的。通过进一步地上层调用，最后发现真正的操作主类 CacheReplicationMonitor，这个类的用途如下：“扫描整个 HDFS 文件系统，根据情况调度块进行缓存。”

在这里笔者做部分补充解释, CacheReplicationMonitor 自身持有一个系统中的标准缓存块列表, 然后通过自身内部的缓存规则, 进行缓存块的添加和移除, 然后对应更新到之前提到过的 pendingCache 和 pendingUncache 列表中, 随后这些信息会被 NameNode 拿来放入回复命令中。这里会有 2 个疑问点:

- ❑ CacheReplicationMonitor 内部维护的系统标准缓存块从哪里来?
- ❑ CacheReplicationMonitor 内部缓存规则、策略是什么, 什么情况下块应该被缓存, 什么情况下又可以取消缓存?

第一个问题会在后面的小节中提到, 这里主要看第二条, 答案在 rescanCached-BlockMap 的方法中。此方法代码处理逻辑比较复杂, 从方法注释中的解释, 我们可以归纳出下面两个基本规则:

- ❑ 任何少于标准副本块个数的副本应该被缓存到新的节点上。
- ❑ 过量副本数的缓存块应该从节点上进行移除。

其实仔细一想, 这个策略还是很巧妙的, 尽量多缓存一些副本数不足的副本(缓存相当于充当了一块副本), 移除掉副本数过多的多余缓存。

2.1.5 系统持有的缓存块列表如何更新

上文中提到过 CacheReplicationMonitor 对象持有的系统缓存块列表如何被更新的问题, 这个列表是用来发送 pendingCache、pendingUncache 信息的基础。因为缓存块列表是系统全局持有的, 会存在反馈上报的过程, 相关代码位于心跳处理代码的附近:

```
...
List<DatanodeCommand> cmds = blockReport();
processCommand(cmds == null ? null : cmds.toArray(new DatanodeCommand [cmds.
    size()]));
// 缓存块的上报
DatanodeCommand cmd = cacheReport();
processCommand(new DatanodeCommand[]{ cmd });
...
```

继续调用到下面这行操作:

```
// 获取到 DataNode 上的缓存块 id 列表
List<Long> blockIds = dn.getFSDataset().getCacheReport(bpid);
```

最后又重新调用到了 FSDatasetCache 中的 getCacheBlocks 方法。至此我们可以发现, 这里形成了一个闭环操作, 最后的缓存操作执行者同样也是缓存块情况的反馈者。CacheReplicationMonitor 属于 CacheManager 对象的内部变量, 会从中拿到缓存块的最新信息。同时这里需要注意一点, CacheManager 自身同样可以主动添加新的目标缓存块, 对应的命令是 hdfs cacheadmin。HDFS 缓存块命令的相关内容在后面的内容中会提及到。

2.1.6 缓存块的使用

这时候重新回头看缓存块的使用，问题就显得比较简单了。缓存块在 ShortCircuit（短路读）的操作中被用到（参见第 6 章 6.2.5 节），代码如下：

```
public void requestShortCircuitFds(final ExtendedBlock blk,
    final Token<BlockTokenIdentifier> token,
    SlotId slotId, int maxVersion, boolean supportsReceiptVerification)
    throws IOException {
    ...
    if (slotId != null) {
        boolean isCached = datanode.data.
            isCached(blk.getBlockPoolId(), blk.getBlockId());
        datanode.shortCircuitRegistry.registerSlot(
            ExtendedBlockId.fromExtendedBlock(blk), slotId, isCached);
        registeredSlotId = slotId;
    }
    fis = datanode.requestShortCircuitFdsForRead(blk, token, maxVersion);
    Preconditions.checkNotNull(fis);
    bld.setStatus(SUCCESS);
    bld.setShortCircuitAccessVersion(DataNode.CURRENT_BLOCK_FORMAT_VERSION);
    ...
}
```

最后给出完整调用流程，如图 2-3 所示。我们可以明显看到中间的一个闭环。

2.1.7 HDFS 缓存相关配置

讲述完关于 HDFS 缓存的原理和内容之后，接下来介绍管控此功能的配置项，如下所示：

```
<property>
  <name>dfs.datanode.max.locked.memory</name>
  <value>0</value>
  <description>
    DataNode 用来缓存块的最大内存空间大小，单位用字节表示。系统变量 RLIMIT_MEMLOCK 至少需要设置得比此配置值要大，否则 DataNode 会出现启动失败的现象。在默认的情况下，此配置值为 0，表明默认关闭内存缓存的功能。
  </description>
</property>
```

这个配置项的名称与实际所使用的功能情况不太一致，有点歧义的感觉，可能叫 dfs.datanode.max.cache.memory 比较好理解一些。这个配置项控制的是下面这个变量，在 FSDataSetCache 构造函数中首先会拿到此属性值：

```
this.maxBytes = dataset.datanode.getDnConf().getMaxLockedMemory();
```

然后在 usedBytes 对象的使用上做限制：

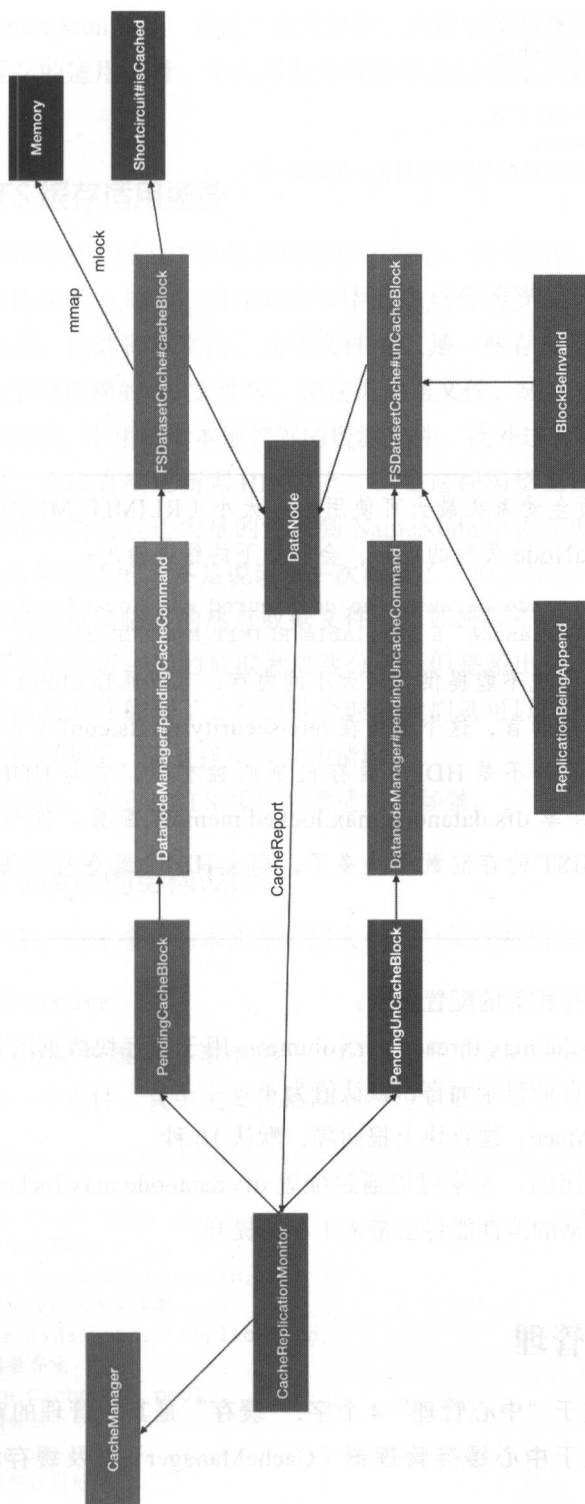


图 2-3 缓存机制流程图

```

211 long reserve(long count) {
    count = rounder.round(count);
    while (true) {
        long cur = usedBytes.get();
        long next = cur + count;
        // 如果当前内存使用量已经超过最大允许使用值, 则返回 -1
        if (next > maxBytes) {
            return -1;
        }
        if (usedBytes.compareAndSet(cur, next)) {
            return next;
        }
    }
}

```



注意

- 第一点, 此配置项会受系统最大可使用内存大小 (RLIMIT_MEMLOCK) 的影响, 造成启动 DataNode 失败的现象, 会有如下的错误输出:

```

Cannot start datanode because the configured max locked memory size... is
more than the datanode's available RLIMIT_MEMLOCK ulimit.

```

上面的记录表明操作系统不能提供相应大小的内存, 可以通过 `ulimit -l <value>` 命令对此进行调整。一般而言, 这个值是在 `/etc/security/limits.conf` 文件中配置的。

- 第二点, 此配置项并不是 HDFS 缓存机制所独有的, 它与 HDFS 的 LAZY_PERSIST 策略会共享 `dfs.datanode.max.locked.memory` 配置。换句话说, 如果用于 LAZY_PERSIST 的存储数据块多了, 那么 HDFS 缓存所能使用的空间自然就会变少。

最后附上几个与 HDFS 缓存相关的配置参数:

- `dfs.datanode.fsdatasetcache.max.threads.per.volume`: 用于缓存块数据的最大线程数, 这个线程数是针对每个存储目录而言, 默认值为 4。
- `dfs.cachereport.intervalMsec`: 缓存块上报间隔, 默认 10 秒。

HDFS 缓存功能默认是关闭的, 大家可以通过配置 `dfs.datanode.max.locked.memory` 的值来开启此功能, 对于文件数据的读性能将会带来不小的提升。

2.2 HDFS 中心缓存管理

HDFS 中心缓存管理偏重于“中心管理”4 个字,“缓存”是其中管理的对象。HDFS 中心缓存管理机制主要依赖于中心缓存管理器 (CacheManager) 以及缓存块监控服务

(CacheReplicationMonitor)。通过二者的协作，来控制集群缓存块的状态。在本节中，我们将从 HDFS 缓存的适用场景、中心缓存管理的原理过程以及 HDFS 缓存的命令使用三个方面展开讲述。

2.2.1 HDFS 缓存适用场景

首先，我们先要了解 HDFS 缓存所适用的场景，换句话说，它能解决哪些具体的问题。

归纳来说是这么一句话：缓存 HDFS 中的热点公共资源文件和短期临时的热点数据文件。第一种场景：公共资源文件。这些文件可以是一些存放于 HDFS 中的依赖资源 jar 包，或是一些算法学习依赖的 .so 文件等。像这类数据文件，放在 HDFS 上的好处是我们可以可以在 HDFS 上全局共享，不用依赖本地机器的资源文件。此外这种做法易于管理，如果想要进行资源包的升级，可以直接更新到 HDFS 上。但是这种场景更好的做法是把它做成分布式缓存，否则在程序中将会发送大量的请求到 NameNode 中去获取这些资源文件的内容。而且这种请求量是非常恐怖的，不是说请求一次就够了，而是使用一次就请求一次。

第二种场景：短期临时的热点数据文件。比如集群中每天运行统计的报表数据，需要读取前一天的或是最近一周的数据做离线分析。但是超出这个期限内的数据基本就很少再用了，就可以视为冷数据了。那么这个时候我们就可以把符合这个时间段的数据做缓存处理，如果数据过期了，就直接从缓存中清除。

以上两种情况，都是 HDFS 缓存非常适用的场景。

2.2.2 HDFS 缓存的结构设计

在 HDFS 中，最终缓存的本质是数据文件。但是在逻辑上，引入了下面几个概念。

1. CacheDirective

CacheDirective 是缓存的基本单元，但是这里 CacheDirective 不一定针对的是一个目录，也可以是一个文件。其中主要包括以下一些变量：

```
public final class CacheDirective implements IntrusiveCollection.Element {
    // 唯一标识 Id
    private final long id;
    // 目标缓存路径
    private final String path;
    // 对应路径的文件副本数
    private final short replication;
    // 所属缓存池
    private CachePool pool;
    // 过期时间
    private final long expiryTime;
    // 相关统计指标
```

```

private long bytesNeeded;
private long bytesCached;
private long filesNeeded;
private long filesCached;
...

```

在这里，我们看到了一个新的概念：缓存池（CachePool），在此可以得出下面一个结论：CacheDirective 属于对应的缓存池。

2. CachePool

下面是 CachePool 概念的定义：

```

public final class CachePool {
    // 缓存池名称
    @Nonnull
    private final String poolName;
    // 所属用户名
    @Nonnull
    private String ownerName;
    // 所属组名
    @Nonnull
    private String groupName;
    // 缓存池权限，分为读（READ）、写（WRITE）、可执行（EXECUTE）
    @Nonnull
    private FsPermission mode;
    // 缓存池最大允许的缓存字节数
    private long limit;
    // 过期时间
    private long maxRelativeExpiryMs;
    // 变量统计相关值
    private long bytesNeeded;
    private long bytesCached;
    private long filesNeeded;
    private long filesCached;
    ...
    // 缓存对象列表
    @Nonnull
    private final DirectiveList directiveList = new DirectiveList(this);
    ...
}

```

我们可以看到，在缓存池中确实维护了一个缓存单元列表。同时这些缓存池被 CacheManager 所掌管，CacheManager 在这里就好比一个总管理者的角色。在 CacheManager 中还运行着一个很重要的服务：缓存副本监控器（CacheReplicationMonitor）。这个监控程序会周期性地扫描当前最新的缓存路径，并分发缓存块到对应的 DataNode 节点上。这个线程服务在后面还会具体提到。HDFS 缓存的总结构关系如图 2-4 所示。

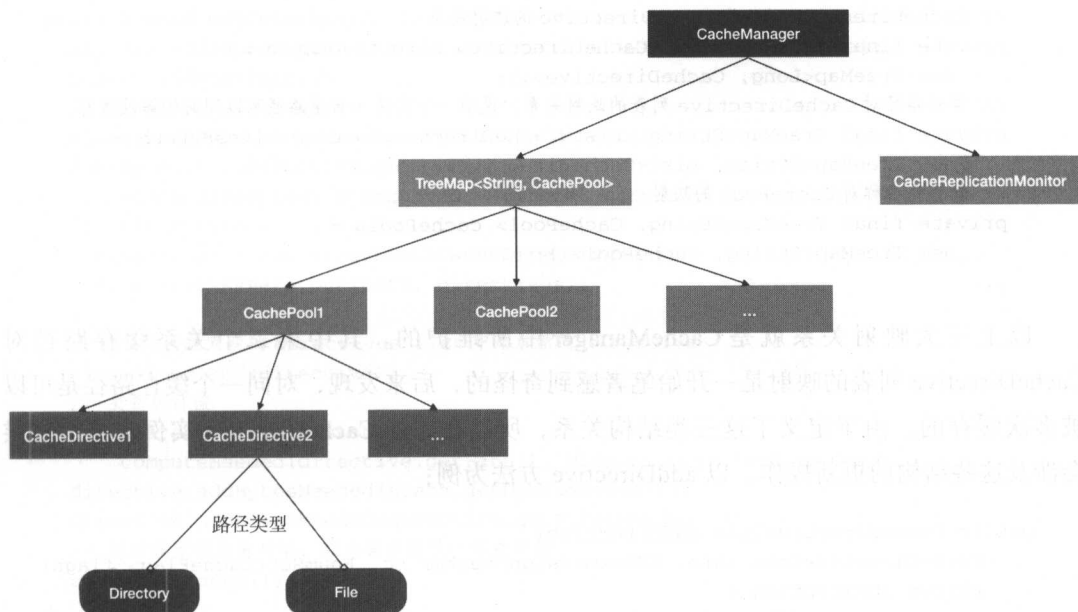


图 2-4 HDFS 缓存总结构关系

2.2.3 HDFS 缓存管理机制分析

在上一节的内容中我们偏重介绍 HDFS 缓存块的内容，而在本节我们会偏向讲述缓存管理方面的内容，主要为以下两方面的内容：

- ❑ CacheAdmin CLI 命令在 CacheManager 的实现。
 - ❑ CacheManager 的 CacheReplicationMonitor 如何将目标缓存文件缓存到 DataNode 中。
- 下面先来看第一点涉及的内容。

1. CacheAdmin CLI 命令在 CacheManager 的实现

CacheAdmin 是 HDFS 中缓存块的管理命令。在 CacheAdmin 中的每个操作命令，最后通过 RPC 调用都会对应到 CacheManager 中的一个具体操作方法。在此过程中要解决下面几个主要疑问点：

- ❑ CacheManager 维护了怎样的 CachePool、CacheDirective 关系？
- ❑ 添加新的 CacheDirective、CachePool 有哪些特殊的细节？

对于第一个问题，CacheManager 确实维护了 CachePool、CacheDirective 的多种映射关系，如下所示：

```
public final class CacheManager {
```

```
...
```

```

// CacheDirective id 对 CacheDirective 的映射关系
private final TreeMap<Long, CacheDirective> directivesById =
    new TreeMap<Long, CacheDirective>();
// 缓存路径对 CacheDirective 列表的映射关系, 说明一个文件 / 目录路径可以同时被多次缓存
private final TreeMap<String, List<CacheDirective>> directivesByPath =
    new TreeMap<String, List<CacheDirective>>();
// 缓存池名称对 CachePool 的映射
private final TreeMap<String, CachePool> cachePools =
    new TreeMap<String, CachePool>();
...

```

以上三大映射关系就是 **CacheManager** 中所维护的。其中第二个关系缓存路径对 **CacheDirective** 列表的映射是一开始笔者感到奇怪的, 后来发现, 对同一个缓存路径是可以被多次缓存的。由于定义了这三类结构关系, 所以在添加 **CacheDirective** 实例对象的时候会涉及这些结构的更新操作。以 **addDirective** 方法为例:

```

public CacheDirectiveInfo addDirective(
    CacheDirectiveInfo info, FSPermissionChecker pc, EnumSet<CacheFlag> flags)
    throws IOException {
    assert namesystem.hasWriteLock();
    CacheDirective directive;
    try {
        // 获取所属缓存池
        CachePool pool = getCachePool(validatePoolName(info));
        // 验证是否有权限
        checkWritePermission(pc, pool);
        // 验证缓存路径
        String path = validatePath(info);
        // 验证副本数
        short replication = validateReplication(info, (short)1);
        // 验证过期时间
        long expiryTime = validateExpiryTime(info, pool.getMaxRelativeExpiryMs());
        // 如果带上了 force 参数, 就要验证 CachePool 是否还有剩余空间添加新的缓存
        if (!flags.contains(CacheFlag.FORCE)) {
            checkLimit(pool, path, replication);
        }
        // 获取下一个 Id
        long id = getNextDirectiveId();
        // 构建新的 CacheDirective 实例
        directive = new CacheDirective(id, path, replication, expiryTime);
        // 进行添加操作
        addInternal(directive, pool);
    } catch (IOException e) {
        ...
    }
}

```

在上面的方法中, 有一个步骤需要注意, 就是 **force** 参数处理时的空间限制处理, 默认情况下 **CachePool** 是不受限的。在 **addInternal** 中会涉及存储关系的更新:

```

private void addInternal(CacheDirective directive, CachePool pool) {
    boolean addedDirective = pool.getDirectiveList().add(directive);
    assert addedDirective
    // 添加新的 id->directive
    directivesById.put(directive.getId(), directive);
    String path = directive.getPath();
    List<CacheDirective> directives = directivesByPath.get(path);
    if (directives == null) {
        directives = new ArrayList<CacheDirective>(1);
        directivesByPath.put(path, directives);
    }
    // 在目标路径对应的 directives list 中添加新的 directive
    directives.add(directive);
    // 更新统计值
    CacheDirectiveStats stats =
        computeNeeded(directive.getPath(), directive.getReplication());
    directive.addBytesNeeded(stats.getBytesNeeded());
    directive.addFilesNeeded(directive.getFilesNeeded());
    // 然后设置需要重扫描, 因为缓存目录已经被更新了
    setNeedsRescan();
}

```

类似的还有添加 **CachePool** 的操作, 代码如下:

```

public CachePoolInfo addCachePool(CachePoolInfo info)
    throws IOException {
    assert namesystem.hasWriteLock();
    CachePool pool;
    try {
        CachePoolInfo.validate(info);
        // 获取缓存池名称
        String poolName = info.getPoolName();
        pool = cachePools.get(poolName);
        // 如果获取到的缓存池不为空, 说明已经存在, 抛异常
        if (pool != null) {
            throw new InvalidRequestException("Cache pool " + poolName
                + " already exists.");
        }
        // 在默认缓存池的基础上构造出新的缓存池对象
        pool = CachePool.createFromInfoAndDefaults(info);
        // 添加到缓存池存储关系列表中
        cachePools.put(pool.getPoolName(), pool);
    } catch (IOException e) {
        LOG.info("addCachePool of " + info + " failed: ", e);
        throw e;
    }
    LOG.info("addCachePool of {} successful.", info);
    return pool.getInfo(true);
}

```

当然在 **CacheManager** 中还有其他修改缓存单元和列表操作的方法, 在逻辑上没有什么

特别之处，这里就不过多地介绍了。写到这里，再回头看 CacheManager 维护的三种存储关系，还是有一定的巧妙之处的，为什么这么说呢？

CacheManager 通过 id 到 CacheDirective，路径到 CacheDirective 列表和名称到 CachePool 的多个映射关系，使得原本逻辑上的父子关系结构平级化了，方便了多条件地灵活查询。比如说我们通过 id 去找对应的缓存对象，就不需要重新遍历查找了。

2. CacheReplicationMonitor 缓存监控服务

如果把上文中 CacheManger 的缓存添加删除操作比喻为一个工厂中的零件加工过程，那么 CacheReplicationMonitor 服务就好比是一个强大的发动机，它会将这些零件完美地处理并分配到对应的车间中去。可以说 CacheReplicationMonitor 服务是一个指挥者的角色。

但是这个“指挥者”也同样被“大管家”CacheManager 所掌管，并控制着它的开启与关闭。

```
public final class CacheManager {
    ...
    // 缓存管理器中的缓存监控服务
    private CacheReplicationMonitor monitor;
```

缓存监控线程，正如其名称所表示的含义：缓存副本块的监控服务。因为是一个监控类型的服务程序，所以里面通常会有循环执行的操作逻辑。在 CacheReplicationMonitor 中，操作的对象是缓存块，CacheReplicationMonitor 类的定义如下：

```
public class CacheReplicationMonitor extends Thread implements Closeable {
    ...
    // 当前需要缓存的缓存块集合
    private final GSet<CachedBlock, CachedBlock> cachedBlocks;
    ...
```

接着进入其中的 run 方法：

```
public void run() {
    long startTimeMs = 0;
    Thread.currentThread().setName("CacheReplicationMonitor(" +
        System.identityHashCode(this) + ")");
    LOG.info("Starting CacheReplicationMonitor with interval " +
        intervalMs + " milliseconds");
    try {
        long curTimeMs = Time.monotonicNow();
        // 主循环
        while (true) {
            lock.lock();
            try {
                // 是否进行下一次 rescan 扫描的判断
                while (true) {
```

```

// 判断此服务是否已被设置为停止状态
if (shutdown) {
    LOG.info("Shutting down CacheReplicationMonitor");
    return;
}
// 判断已扫描完成的块数是否小于需要完成的数目, 如果是则需要重新进行 rescan
if (completedScanCount < neededScanCount) {
    LOG.debug("Rescanning because of pending operations");
    break;
}
long delta = (startTimeMs + intervalMs) - curTimeMs;
if (delta <= 0) {
    LOG.debug("Rescanning after {} milliseconds", (curTimeMs - startTimeMs));
    break;
}
doRescan.await(delta, TimeUnit.MILLISECONDS);
curTimeMs = Time.monotonicNow();
}
} finally {
    lock.unlock();
}
startTimeMs = curTimeMs;
mark = !mark;
// 执行新的扫描操作
rescan();
...
}

```

这里 `rescan` 方法才是我们最关注的, 此方法的代码如下所示:

```

private void rescan() throws InterruptedException {
    scannedDirectives = 0;
    scannedBlocks = 0;
    try {
        ...
        // 重置统计计数值
        resetStatistics();
        // 重新扫描缓存目录单元
        rescanCacheDirectives();
        // 重新扫描当前缓存块
        rescanCachedBlockMap();
        blockManager.getDataNodeManager().resetLastCachingDirectiveSentTime();
    } finally {
        namesystem.writeUnlock();
    }
}

```

在 `rescan` 内部, 会进行三部分操作。

第一部分, `resetStatistics` 重置统计变量计数值。因为要进行完全新一轮的缓存过程, 所以 `CachePool` 以及其所包含的 `CacheDirective` 都要重新计数, 代码执行过程如下:

```

private void resetStatistics() {
    for (CachePool pool: cacheManager.getCachePools()) {
        // 对每个 cachePool 进行重置计数
        pool.resetStatistics();
    }
    for (CacheDirective directive: cacheManager.getCacheDirectives()) {
        // 对每个 CacheDirective 进行重置计数
        directive.resetStatistics();
    }
}

```

第二部分，rescanCacheDirectives。在这个过程中会扫描之前保存在 CacheManager 中的那些 CacheDirectives。具体操作如下：

```

private void rescanCacheDirectives() {
    FSDirectory fsDir = namesystem.getFSDirectory();
    final long now = new Date().getTime();
    // 获取缓存管理器中保存的全部缓存单元对象
    for (CacheDirective directive : cacheManager.getCacheDirectives()) {
        scannedDirectives++;
        // 跳过已经过期的缓存单元
        if (directive.getExpiryTime() > 0 && directive.getExpiryTime() <= now) {
            LOG.debug("Directive {}: the directive expired at {} (now = {})",
                directive.getId(), directive.getExpiryTime(), now);
            continue;
        }
        String path = directive.getPath();
        INode node;
        try {
            // 获取缓存单元中缓存路径所代表的 INode 对象
            node = fsDir.getINode(path);
        } catch (UnresolvedLinkException e) {
            // We don't cache through symlinks
            LOG.debug("Directive {}: got UnresolvedLinkException while resolving "
                + "path {}", directive.getId(), path);
            continue;
        }
        if (node == null) {
            LOG.debug("Directive {}: No inode found at {}", directive.getId(),
                path);
        } else if (node.isDirectory()) {
            // 如果此路径代表的是目录
            INodeDirectory dir = node.asDirectory();
            ReadOnlyList<INode> children = dir
                .getChildrenList(Snapshot.CURRENT_STATE_ID);
            for (INode child : children) {
                if (child.isFile()) {
                    // 则扫描文件
                    rescanFile(directive, child.asFile());
                }
            }
        }
    }
}

```



```

    }
}
} else if (node.isFile()) {
    // 如果是文件则直接进行扫描
    rescanFile(directive, node.asFile());
} else {
    LOG.debug("Directive {}: ignoring non-directive, non-file inode {} ",
        directive.getId(), node);
}
}
}
}

```

这里继续进入 `rescanFile` 的操作方法:

```

private void rescanFile(CacheDirective directive, INodeFile file) {
    // 获取文件所包含的块信息
    BlockInfo[] blockInfos = file.getBlocks();

    // 增加文件需要数的计数统计
    directive.addFilesNeeded(1);
    // 计算缓存需要的字节大小, 不包括正在被写的块
    long neededTotal = file.computeFileSizeNotIncludingLastUcBlock() *
        directive.getReplication();
    directive.addBytesNeeded(neededTotal);

    // 获取此存储对象的所属缓存池
    CachePool pool = directive.getPool();
    // 如果缓存池所需要缓存的空间大小超过限制, 则返回
    if (pool.getBytesNeeded() > pool.getLimit()) {
        LOG.debug("Directive {}: not scanning file {} because " +
            "bytesNeeded for pool {} is {}, but the pool's limit is {}",
            directive.getId(),
            file.getFullPathName(),
            pool.getPoolName(),
            pool.getBytesNeeded(),
            pool.getLimit());
        return;
    }

    long cachedTotal = 0;
    // 遍历目标缓存文件所拥有的块
    for (BlockInfo blockInfo : blockInfos) {
        if (!blockInfo.getBlockUCState().equals(BlockUCState.COMPLETE)) {
            // 这里不缓存正在被写入数据的块
            LOG.trace("Directive {}: can't cache block {} because it is in state "
                + "{}", not COMPLETE.", directive.getId(), blockInfo,
                blockInfo.getBlockUCState());
        };
        continue;
    }
    // 构造缓存块
}

```

```

Block block = new Block(blockInfo.getBlockId());
CachedBlock ncblock = new CachedBlock(block.getBlockId(),
    directive.getReplication(), mark);
CachedBlock ocblock = cachedBlocks.get(ncblock);
if (ocblock == null) {
    // 如果目标缓存块列表中不存在当前缓存块, 则进行添加
    cachedBlocks.put(ncblock);
    ocblock = ncblock;
} else {
    // 如果存在, 则进行相关变量的更新
    ...
    ocblock.setReplicationAndMark(directive.getReplication(), mark);
}
}
LOG.trace("Directive {}: setting replication for block {} to {}",
    directive.getId(), blockInfo, ocblock.getReplication());
}
...
}

```

可以将上述操作归纳成下面几个过程:

- 1) 获取缓存文件所拥有的块组信息。
- 2) 判断 CachePool 的缓存大小是否超过限制。
- 3) 遍历块, 并根据块构造缓存块。
- 4) 如果缓存块列表中不存在当前构造的缓存块则直接添加, 否则进行部分信息的更新。

可能这里有人会有疑问, 为什么缓存块列表内已经包含了目标缓存块呢? 有两类情况会导致此现象的发生:

- ❑ 第一种情况: 当前目标缓存块在上一轮由于种种条件限制, 没有被缓存出去, 所以就没有被移除掉。
- ❑ 第二种情况: 此缓存块已经被缓存到 DataNode 上, 后来经过 DataNode 的缓存块汇报操作又上报到 CacheManager 的缓存列表中。而 CacheReplicationMonitor 处理的正是 CacheManager 中的缓存块列表。

既然缓存块列表已经包含了目标缓存的块了, 是否会造成块的重复缓存, 从而造成内存空间的浪费呢? 在下面 rescanCachedBlockMap 的操作中, 我们将寻找到答案。

第三部分: rescanCachedBlockMap 操作。此过程是 resan 内部 3 个方法中逻辑最为复杂的操作, 下面进行分段分析。

首先会做一些前期的操作, 代码如下:

```

private void rescanCachedBlockMap() {
    // 遍历目标缓存块列表, 移除 DataNode 中那些会耗尽 DataNode 缓存空间的待缓存块
    for (Iterator<CachedBlock> cbIter = cachedBlocks.iterator();
        cbIter.hasNext(); ) {

```

```

scannedBlocks++;
CachedBlock cblock = cbIter.next();
// 获取不同缓存状态的缓存块中的节点列表
List<DatanodeDescriptor> pendingCached =
    cblock.getDatanodes(Type.PENDING_CACHED);
List<DatanodeDescriptor> cached =
    cblock.getDatanodes(Type.CACHED);
List<DatanodeDescriptor> pendingUncached =
    cblock.getDatanodes(Type.PENDING_UNCACHED);
...

```

然后根据上述当前缓存块的不同缓存状态的信息，来计算当前块的缓存数目信息，进行下面 2 部分的处理。

```

...
// 如果当前缓存数已经满足需要缓存的数量，则移除当前待缓存块
if (numCached >= neededCached) {
    for (Iterator<DatanodeDescriptor> iter = pendingCached.iterator();
         iter.hasNext(); ) {
        DatanodeDescriptor datanode = iter.next();
        datanode.getPendingCached().remove(cblock);
        iter.remove();
        LOG.trace("Block {}: removing from PENDING_CACHED for node {} "
            + "because we already have {} cached replicas and we only " +
            "need {}",
            cblock.getBlockId(), datanode.getDatanodeUuid(), numCached,
            neededCached
        );
    }
}
// 如果当前缓存块还未达到目标需要的缓存数，则移除待取消缓存的块
if (numCached < neededCached) {
    for (Iterator<DatanodeDescriptor> iter = pendingUncached.iterator();
         iter.hasNext(); ) {
        DatanodeDescriptor datanode = iter.next();
        datanode.getPendingUncached().remove(cblock);
        iter.remove();
        LOG.trace("Block {}: removing from PENDING_UNCACHED for node {} "
            + "because we only have {} cached replicas and we need " +
            "{}", cblock.getBlockId(), datanode.getDatanodeUuid(),
            numCached, neededCached
        );
    }
}
}

```

然后添加还需要缓存的块或还需要取消缓存的块：

```

if (neededUncached > 0) {
    // 添加需要取消缓存的块
    addNewPendingUncached(neededUncached, cblock, cached,

```

```

        pendingUncached);
    } else {
        int additionalCachedNeeded = neededCached -
            (numCached + pendingCached.size());
        // 同时添加剩余数目的缓存块到待缓存列表中
        if (additionalCachedNeeded > 0) {
            addNewPendingCached(additionalCachedNeeded, cblock, cached,
                pendingCached);
        }
    }
}

```

如果前面的条件都已经满足了，则当前正在进行遍历处理的目标缓存块会被移除掉，表明此块已经成功被缓存到 DataNode 节点上了。

```

// 如果任何条件都满足了，则对目标缓存块进行移除
if ((neededCached == 0) &&
    pendingUncached.isEmpty() &&
    pendingCached.isEmpty()) {
    // we have nothing more to do with this block.
    LOG.trace("Block {}: removing from cachedBlocks, since neededCached "
        + " == 0, and pendingUncached and pendingCached are empty.",
        cblock.getBlockId());
};
cbIter.remove();
}

```

以上内容的核心点在于变量值 `neededCached`，而这个值本质上就是目标缓存块的自身副本数。所以学习完这部分的过程，之前提出的重复缓存的问题自然也就解决了。对于同一个缓存块，CacheReplicationMonitor 服务是不会进行多余的缓存动作。图 2-5 是 `rescan` 方法的过程图。

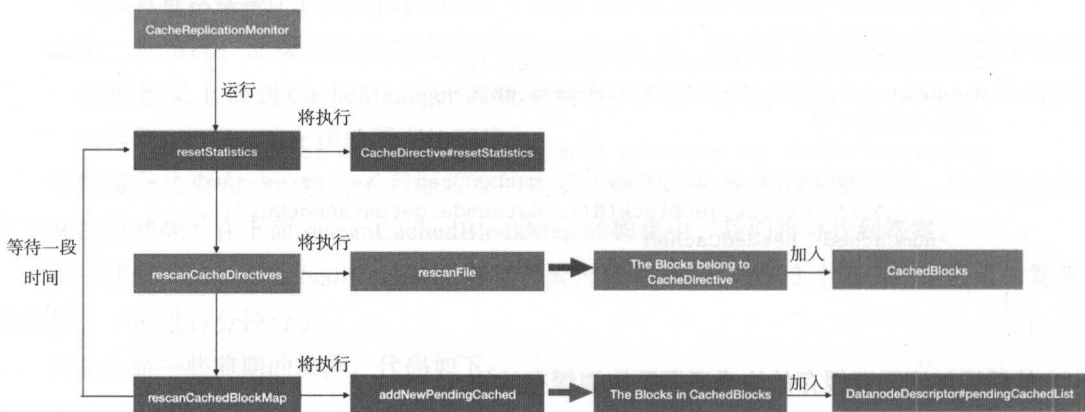


图 2-5 rescan 过程

2.2.4 HDFS 中心缓存疑问点

在 HDFS 缓存逻辑的实现过程中, 笔者认为有几处地方在实现上存在问题, 主要有下面 2 个疑问点:

第一个疑问点, CacheManager 的 checkLimit 方法在运行的时候默认副本数总是为 1。在 checkLimit 方法的前面部分用副本数计算字节需求量:

```
private void checkLimit(CachePool pool, String path,
    short replication) throws InvalidRequestException {
    // 副本数作为参数传入, 用以计算统计值, 但是没有被用上
    CacheDirectiveStats stats = computeNeeded(path, replication);
    ...
}
```

但其实在 computeNeeded 方法中副本数并没有被用上:

```
private CacheDirectiveStats computeNeeded(String path, short replication) {
    ...
    return new CacheDirectiveStats.Builder()
        // 在这个地方理应是 requestedBytes * replication
        .setBytesNeeded(requestedBytes)
        .setFilesCached(requestedFiles)
        .build();
}
```

在上面注释的地方如果没有用上副本数, 就会导致后面异常时输出的信息有误。执行过程如下:

```
private void checkLimit(CachePool pool, String path,
    short replication) throws InvalidRequestException {
    // 副本数作为参数传入, 用以计算统计值, 但是没有被用上
    CacheDirectiveStats stats = computeNeeded(path, replication);
    if (pool.getLimit() == CachePoolInfo.LIMIT_UNLIMITED) {
        return;
    }
    // 因为副本数没用上, 在这里需要乘上副本系数
    if (pool.getBytesNeeded() + (stats.getBytesNeeded() * replication) > pool
        .getLimit()) {
        // 但是在下面的输出中又除了副本数, 因此副本数并没有乘上
        throw new InvalidRequestException("Caching path " + path + " of size "
            + stats.getBytesNeeded() / replication + " bytes at replication "
            + replication + " would exceed pool " + pool.getPoolName()
            + "'s remaining capacity of "
            + (pool.getLimit() - pool.getBytesNeeded()) + " bytes.");
    }
}
```

这个问题已确认是一个 bug, 在社区上已有相应 JIRA: HDFS-10448 (CacheManager#checkLimit always assumes a replication factor of 1)。

第二个疑问点，CacheReplicationMonitor 在缓存目录的时候没有考虑到子目录的情况，只是处理了直接孩子文件的情况，代码如下：

```
private void rescanCacheDirectives() {
    FSDirectory fsDir = namesystem.getFSDirectory();
    final long now = new Date().getTime();
    // 遍历缓存管理器中需要缓存的基本单元
    for (CacheDirective directive : cacheManager.getCacheDirectives()) {
        //...
        if (node == null) {
            LOG.debug("Directive {}: No inode found at {}", directive.getId(),
                path);
        } else if (node.isDirectory()) {
            // 如果此对象包含的 INode 是目录的话，则遍历孩子节点
            INodeDirectory dir = node.asDirectory();
            ReadOnlyList<INode> children = dir
                .getChildrenList(Snapshot.CURRENT_STATE_ID);
            for (INode child : children) {
                if (child.isFile()) {
                    rescanFile(directive, child.asFile());
                }
                // 这里缺少了孩子是目录的情况
            }
        } else if (node.isFile()) {
            // 如果此对象包含的 INode 是纯文件，则直接进行处理
            rescanFile(directive, node.asFile());
        } else {
            LOG.debug("Directive {}: ignoring non-directive, non-file inode {} ",
                directive.getId(), node);
        }
    }
}
```

后来笔者看了官网的介绍，目前的版本中只对缓存路径下的第一层孩子做处理，并没有做到对路径的递归处理。递归处理缓存路径是对这种情况的一个优化，目前社区已有相应的 JIRA 来完善这一点：HDFS-10594 (HDFS-4949 should support recursive cache directives)，感兴趣的同学可以关注一下这个 JIRA。

2.2.5 HDFS CacheAdmin 命令使用

最后来介绍 HDFS 中用于缓存块操作的相关命令使用。这些命令都集中在 hdfs cacheadmin 命令下，在 Hadoop 客户端中输入如下指令，就会弹出所有使用命令：

```
$ hdfs cacheadmin
Usage: bin/hdfs cacheadmin [COMMAND] // 以下为 hdfs cacheadmin 的所有使用命令
[-addDirective -path <path> -pool <pool-name> [-force] [-replication
<replication>] [-ttl <time-to-live>]] // 添加缓存单元命令
```

```

[-modifyDirective -id <id> [-path <path>] [-force] [-replication
  <replication>] [-pool <pool-name>] [-ttl <time-to-live>]] // 修改缓
  存单元命令
[-listDirectives [-stats] [-path <path>] [-pool <pool>] [-id <id>] //
  列出满足条件的缓存单元列表
[-removeDirective <id>] // 删除指定 id 对应的缓存单元
[-removeDirectives -path <path>] // 删除指定路径对应的缓存单元
[-addPool <name> [-owner <owner>] [-group <group>] [-mode <mode>]
  [-limit <limit>] [-maxTtl <maxTtl>] // 添加新的缓存池
[-modifyPool <name> [-owner <owner>] [-group <group>] [-mode <mode>]
  [-limit <limit>] [-maxTtl <maxTtl>]] // 修改缓存池
[-removePool <name>] // 删除指定名称缓存池
[-listPools [-stats] [<name>]] // 列出满足条件的缓存池
[-help <command-name>] // 查阅具体某条命令的使用帮助信息

```

以上命令中除了最后一个 -help 帮助命令之外，其余 9 个命令都与缓存操作相关。在这些命令中，还可以划分为两大类：一类是 CachePool 相关命令，另一类是 CacheDirective 相关命令。分类图如图 2-6 所示。

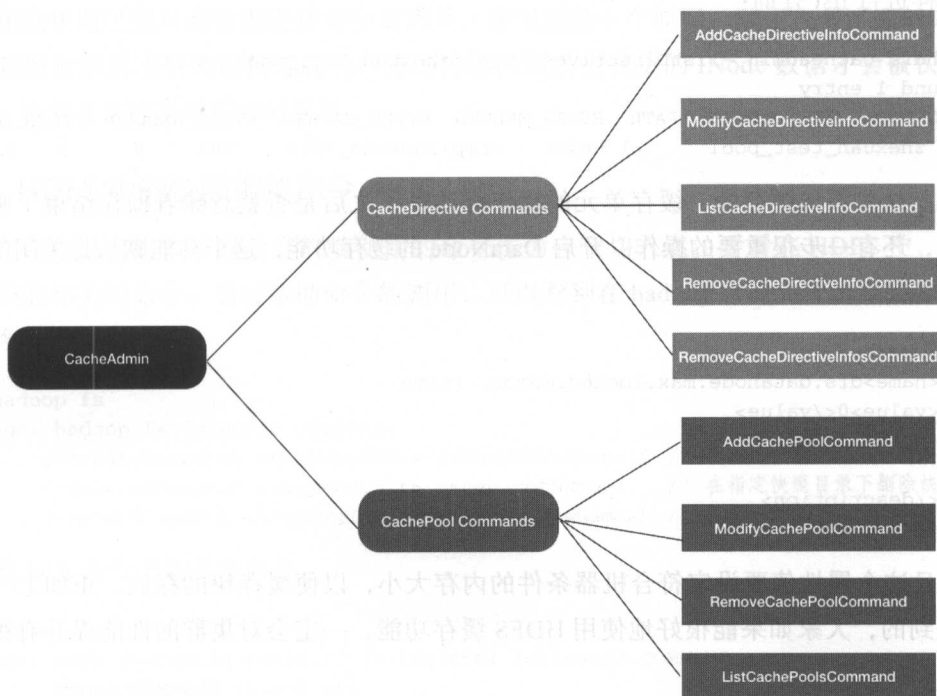


图 2-6 HDFS CacheAdmin 相关命令

下面演示一下笔者在测试集群中的操作结果。

首先，我们需要新建一个缓存池：

```
$ hdfs cacheadmin -addPool zhexuan_test_pool
Successfully added cache pool zhexuan_test_pool.
```

然后用 listPool 命令显示一下是否创建成功：

```
$ hdfs cacheadmin -listPools
Found 1 result.
NAME          OWNER  GROUP  MODE          LIMIT  MAXTTL
zhexuan_test_pool  data   data   rwxr-xr-x     unlimited  never
```

这里需要挑选一个目标缓存的文件或目录，比如下面这个临时文件：

```
-rwx    2 zhexuan supergroup      781 2016-04-15 10:51 /tmp/zhexuan_file
```

调用 addCacheDirective 命令并带上必要的参数，然后加入到刚刚建好的 test_pool 缓存池中：

```
$ hdfs cacheadmin -addDirective -path /tmp/zhexuan_file -pool zhexuan_test_pool
Added cache directive 1
```

同样进行 list 查询：

```
$ hdfs cacheadmin -listDirectives -pool zhexuan_test_pool -stats
Found 1 entry
ID POOL  REPL EXPIRY  PATH  BYTES_NEEDED  BYTES_CACHED  FILES_NEEDED  FILES_CACHED
1 zhexuan_test_pool      1 never   /tmp/zhexuan_file    781    0      1    0
```

以上这些添加缓存池、缓存单元的操作都结束了之后是否就意味着操作结束了呢？并非如此，还有一步很重要的操作：开启 DataNode 的缓存功能，这个功能默认是关闭的。开启该功能需要配置此属性值：

```
<property>
  <name>dfs.datanode.max.locked.memory</name>
  <value>0</value>
  <description>
    ...
  </description>
</property>
```

而且这个属性值要设定符合机器条件的内存大小，以便缓存块的存放。正如上一节末尾所提到的，大家如果能很好地使用 HDFS 缓存功能，一定会对集群的性能提升有很大的帮助。

2.3 HDFS 快照管理

之前章节中讲述了几个最近几年 HDFS 中比较重要的特性（比如异构存储），本节继续

探讨另外一个重要特性：**快照 (Snapshot)**。快照是一个非常好的东西，快照就好比拍风景照时的那一个瞬间的投影，过了那个时间点之后，又会有新的一个投影。所以快照用一个更好的词来形容是“瞬间映像”。在本节中，我们要关注的焦点是 HDFS 的快照，快照在 HDFS 中的实现有其独特之处。在下文中，我们将主要讲述快照的概念、HDFS 快照的管理以及快照的使用三个方面的内容。

2.3.1 快照概念

在分析 HDFS 内部的快照管理之前，需要先了解快照的概念。首先有一个根本的原则：“快照不是数据的简单拷贝，快照只做差异的记录。”

这一原则在其他很多系统的快照概念中都是适用的，比如磁盘快照，也是不保存真实数据的。因为不保存实际的数据，所以快照的生成往往非常迅速。在 HDFS 中，如果在其中一个目录比如 /A 下创建一个快照，则快照文件中将会存在与 /A 目录下完全一致的子目录文件结构以及相应的属性信息，通过 `fs -cat` 命令也能看到里面具体的文件内容。但是这并不意味着快照已经对此数据进行完全的拷贝。这里遵循一个原则：对于大多不变的数据，你所看到的数据其实是当前物理路径所指的内容，而发生变更的 INode 数据才会被快照额外拷贝，也就是前面所说的差异拷贝。

2.3.2 HDFS 中的快照相关命令

我们首先以 HDFS 暴露给客户端使用的命令作为一个切入点，看看在 HDFS 中，存在哪些与快照相关的命令。通过帮助命令的调用，可以看到在 `hadoop fs` 命令下存在以下快照操作命令：

```
$ hadoop fs
Usage: hadoop fs [generic options]
    [-createSnapshot <snapshotDir> [<snapshotName>]] // 在指定快照目录下创建快照
    [-deleteSnapshot <snapshotDir> <snapshotName>]   // 在指定快照目录下删除快照
    [-renameSnapshot <snapshotDir> <oldName> <newName>] // 在指定快照目录下重命名某快照
```

还有 `hdfs` 命令下的几个命令：

```
$ hdfs
Usage: hdfs [--config confdir] [--loglevel loglevel] COMMAND
    where COMMAND is one of:
    snapshotDiff          // 比较两个快照之间的不同或是比较当前内容与某快照之间的不同
    lsSnapshottableDir    // 列出所属当前用户的所有的快照目录
```

以上两部分总共包含了 6 个客户端命令，通过命令的名称以及对应的操作解释，我们大概能明白其作用。这些命令的具体使用方法不是本节内容的重点，具体用法可查阅 Hadoop 官方文档。

如果大家仔细观察上述的 6 个命令，可以看出其中主要围绕着两个概念：

- 快照目录 (Snapshottable Directories)。
- 具体快照 (Snapshot)。

上述两个概念在逻辑上的关系如下：一个快照目录下可以有多个快照文件，快照目录可以创建、删除自身目录下的快照文件，同时快照目录本身又被快照目录管理器所管理。

这里面就引出了更深层次的内容：HDFS 内部的快照管理机制。

2.3.3 HDFS 内部的快照管理机制

1. 快照结构关系

下面我们从源码层面来分析上文提到的对应关系。

1) 快照管理器管理多个快照目录：

```
public class SnapshotManager implements SnapshotStatsMXBean {
    ...
    // 快照目录映射图
    private final Map<Long, INodeDirectory> snapshottables =
        new HashMap<Long, INodeDirectory>();
    ...
}
```

其实每个快照目录就是我们非常熟悉的 INodeDirectory 类。

2) 一个快照目录拥有多个快照文件。快照在快照目录中的存放不是很明显，它是作为一个额外属性存在于 INodeDirectory 的父类 INodeWithAdditionalFields 中。INodeWithAdditionalFields 内部存放有基本的一些变量属性，例如名称、权限、最近修改时间等等，代码中的定义如下：

```
public abstract class INodeWithAdditionalFields extends INode
    implements LinkedElement {
    ...
    private static final Feature[] EMPTY_FEATURE = new Feature[0];
    protected Feature[] features = EMPTY_FEATURE;
    ...
}
```

快照列表存在于一个叫 DirectorySnapshottableFeature 的 Feature 继承类中，源码中的定义如下：

```
public class DirectorySnapshottableFeature extends DirectoryWithSnapshotFeature {
    ...
    // 快照实例列表
    private final List<Snapshot> snapshotsByNames = new ArrayList<Snapshot>();
    ...
}
```

图 2-7 显示了前面提到的两大存放关系，这是本节内容的一个大背景。

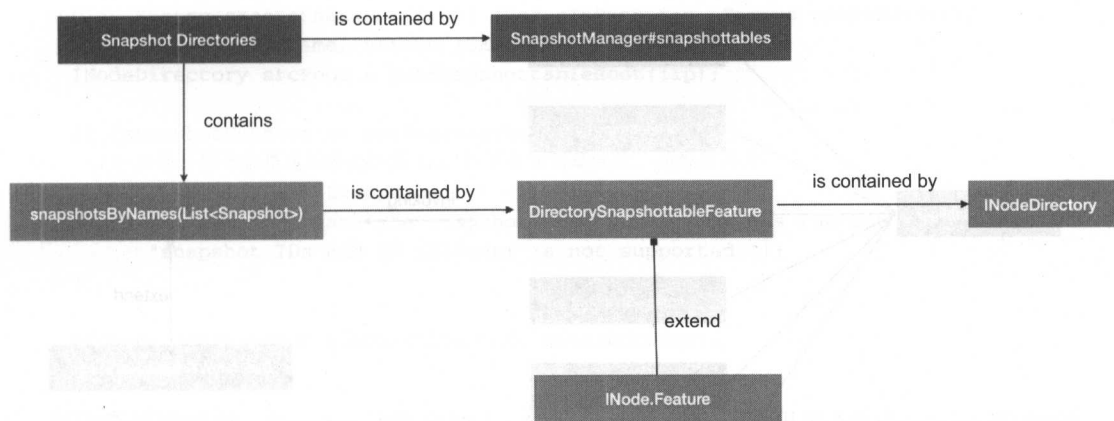


图 2-7 快照结构

2. 快照调用流程

接下来我们来学习快照的调用过程，整个过程以快照管理器（SnapshotManager）作为处理中心。SnapshotManager 负责接收快照操作请求，继而调用相关类进行处理。这里的相关类是 INodeDirectory 中的 Feature 继承类。所以全部过程分为如下两部分：

1) 上游请求的接收，如图 2-8 所示。

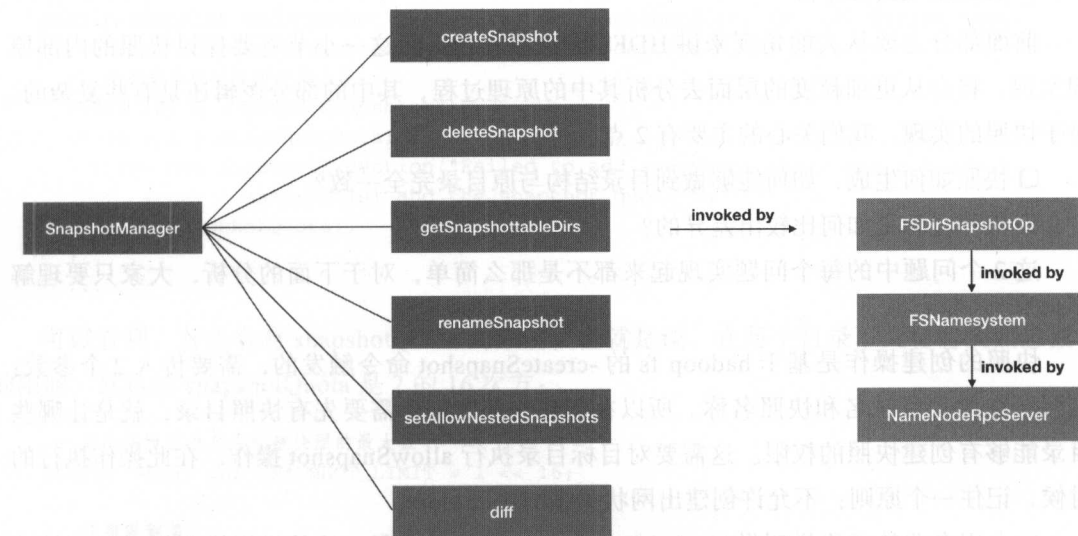


图 2-8 快照的上游调用

2) 请求的下游处理，如图 2-9 所示。

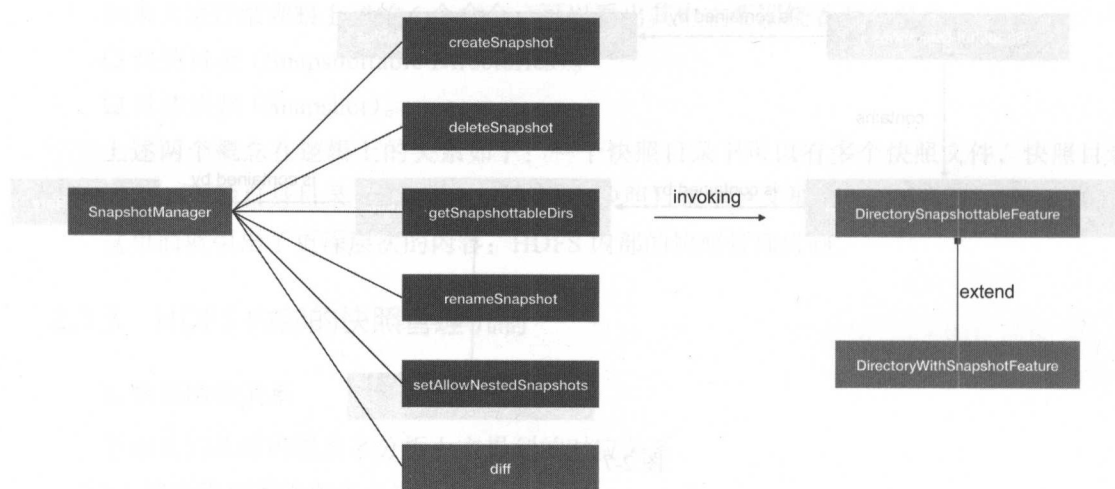


图 2-9 快照的下游处理

在上游请求接收阶段中，其接收方式与以往直接接收 NameNode RPC 请求的方式略有不同，中间还经过了一层 FSDirSnapshotOp 类。在这个类中调用了 SnapshotManager 的操作方法。这样做还是有好处的，可以在 FSNamesystem 的众多操作里很好地辨别和区分操作的类型。

3. 快照原理实现分析

前面部分主要从大的角度来讲 HDFS 的快照管理，而这一小节将要探讨快照的内部原理实现，将会从更细粒度的层面去分析其中的原理过程，其中的部分逻辑还是有些复杂的。对于快照的实现，我们关心的主要有 2 点：

- 快照如何生成，如何能够做到目录结构与原目录完全一致？
- 快照之间是如何比较出差异的？

这 2 个问题中的每个问题实现起来都不是那么简单，对于下面的分析，大家只要理解就行了。

快照的创建操作是基于 `hadoop fs` 的 `-createSnapshot` 命令触发的，需要传入 2 个参数：快照所在的父目录名和快照名称。所以在创建快照之前，需要先有快照目录，就是让哪些目录能够有创建快照的权限。这需要对目标目录执行 `allowSnapshot` 操作。在此操作执行的时候，记住一个原则：不允许创建出网状关系的快照目录。

这个用专业的术语说叫做 `NestedSnapshots`。通俗的解释，就是目标快照目录的子目录和父目录不能够同样为快照目录。这个操作常常容易被用户所忽略。

在进入最终的 `createSnapshot` 方法之前，会做一个系统全局快照数的判断：

```

public String createSnapshot(final INodesInPath iip, String snapshotRoot,
    String snapshotName) throws IOException {
    INodeDirectory srcRoot = getSnapshottableRoot(iip);

    if (snapshotCounter == getMaxSnapshotID()) {
        // 如果快照计数值达到最大快照 ID, 则不允许创建快照, 并抛出异常
        throw new SnapshotException(
            "Failed to create the snapshot. The FileSystem has run out of " +
            "snapshot IDs and ID rollover is not supported.");
    }

    srcRoot.addSnapshot(snapshotCounter, snapshotName);
    ...
}

```

每次新增快照时, Counter 计数会加 1, 然后做计数判断, 这里的 MaxSnapshotID 指的是上限值:

```

public int getMaxSnapshotID() {
    return ((1 << SNAPSHOT_ID_BIT_WIDTH) - 1);
}

```

SNAPSHOT_ID_BIT_WIDTH 值是 24, 所以最大的快照数是 2 的 24 次方减 1, 基本可以认为是用不完的。

继续进入 createSnapshot 方法内部:

```

// 新增快照方法
public Snapshot addSnapshot(INodeDirectory snapshotRoot, int id, String name)
    throws SnapshotException, QuotaExceededException {
    // 检查是否已达快照数量上限
    final int n = getNumSnapshots();
    if (n + 1 > snapshotQuota) {
        throw new SnapshotException("Failed to add snapshot: there are already "
            + n + " snapshot(s) and the snapshot quota is "
            + snapshotQuota);
    }
    ...
}

```

可以看到, 首先会有 snapshotQuota 的限制, 也就是说, 在每个目录下又会有快照总数的限制。默认的 snapshotQuota 是 2 的 16 次方:

```

// 单个快照目录允许创建快照的最大数
static final int SNAPSHOT_LIMIT = 1 << 16;
...
// 快照限制值
private int snapshotQuota = SNAPSHOT_LIMIT;

```

接下来是判断处理:

```

...

```

```

// 新建快照对象类
final Snapshot s = new Snapshot(id, name, snapshotRoot);
final byte[] nameBytes = s.getRoot().getLocalNameBytes();
// 检查是否已经存在同名的快照, 如果有则抛异常
final int i = searchSnapshot(nameBytes);
if (i >= 0) {
    throw new SnapshotException("Failed to add snapshot: there is already a "
        + "snapshot with the same name \"" + Snapshot.getSnapshotName(s) + "\".");
}

// 将此目录的改动记录加入到 diff 列表中
final DirectoryDiff d = getDiffs().addDiff(id, snapshotRoot);
d.setSnapshotRoot(s.getRoot());
snapshotsByNames.add(-i - 1, s);

// 更新快照目录的最近修改时间
final long now = Time.now();
snapshotRoot.updateModificationTime(now, Snapshot.CURRENT_STATE_ID);
s.getRoot().setModificationTime(now, Snapshot.CURRENT_STATE_ID);
return s;
}

```

以下这行操作, 会在快照目录下的隐藏目录 `./snapshot` 下创建目标快照:

```
final Snapshot s = new Snapshot(id, name, snapshotRoot);
```

目标快照是创建好了, 但是另外一个问题来了, 快照是如何完全一致地反映出那一时刻的文件目录信息呢? 而且随着时间的推移, 文件目录早已发生了改动, 快照还是能够保存当时时刻的元信息数据以及对应那个时间点的真实数据, 这一点也是我们所要关注的。

一种很自然联想到的想法是 HDFS 对当时的文件目录的元信息做了一份拷贝。尽管这能解释前面的说法, 但是如果这种假设成立的话, 会衍生出一个很大的问题。当快照目录下的数据文件在没有做任何变动的情况下, 过多的快照创建无疑是毫无意义并且是浪费空间的, 显然在 HDFS 中不会这么做。所以快照是文件目录元信息的简单拷贝是不完全正确的。

那么 HDFS 到底是怎么做的呢? 官方的注释为: “如果当前快照 id 不是 `Snapshot.CURRENT_STATE_ID`, 则从对应的快照中获取结果, 否则从当前的目录中获取结果。”

`CURRENT_STATE_ID` 的意思是当前快照的状态与当前目录完全一致, 没有发生过文件目录信息的变动。所以在这种情况下, 完全返回当前目录信息即可, 换句话说, `Snapshot.CURRENT_STATE_ID` 的意思就是返回当前状态的目录信息。否则从快照 Id 对应的快照中获取结果。从这段注释中可以提取出以下关键信息:

每个快照都有对应自身目录下的 `Inode` 信息列表, 以快照 Id 作为区分标识。

以上推论可以从获取子目录节点的 `getChildrenList` 方法中进行查阅:

```
public ReadOnlyList<Inode> getChildrenList(final int snapshotId) {
```

```

DirectoryWithSnapshotFeature sf;
// 如果当前快照 Id 是 CURENF_STATE_ID 或当前目录的快照特性为空, 则直接返回当前目录的子节点
// 信息
if (snapshotId == Snapshot.CURRENT_STATE_ID
    || (sf = this.getDirectoryWithSnapshotFeature()) == null) {
    return getCurrentChildrenList();
}
// 否则从对应的快照中返回信息
return sf.getChildrenList(this, snapshotId);
}

```

sf.getChildrenList 方法的源代码如下:

```

public ReadOnlyList<INode> getChildrenList(INodeDirectory currentINode,
    final int snapshotId) {
    // 根据快照 Id 取出对应目录的变更对象信息
    final DirectoryDiff diff = diffs.getDiffById(snapshotId);
    // 如果变更目录对象为空则直接返回当前目录的孩子信息, 否则从变更对象中获取子节点列表
    return diff != null ? diff.getChildrenList(currentINode) : currentINode
        .getChildrenList(Snapshot.CURRENT_STATE_ID);
}

```

最后一层是通过 (变更) 信息获取子节点信息列表的方法:

```

private ReadOnlyList<INode> getChildrenList(final INodeDirectory currentDir) {
    return new ReadOnlyList<INode>() {
        private List<INode> children = null;

        private List<INode> initChildren() {
            if (children == null) {
                // 获取变更的孩子信息
                final ChildrenDiff combined = new ChildrenDiff();
                for (DirectoryDiff d = DirectoryDiff.this; d != null;
                    d = d.getPosterior()) {
                    combined.combinePosterior(d.diff, null);
                }
                // 与当前的目录 INode 信息融合, 构成新的子节点列表
                children = combined.apply2Current(ReadOnlyList.Util.asList(
                    currentDir.getChildrenList(Snapshot.CURRENT_STATE_ID)));
            }
            return children;
        }
    };
}

```

到这里操作结束, 我们终于知道最终的孩子列表是通过将 diff 发生过变更的 INode 信息与原目录节点信息进行结合, 然后将一个新的子节点信息作为最终结果返回。diff 中保留的 INode 就是当时快照创建时的 INode 信息。

现在再仔细总结一下整个逻辑:

1) HDFS 中只为每个快照保存相对当时快照创建时间点发生过变更的 INode 信息, 只

是“存不同”。

2) 获取快照信息时, 根据快照 Id 和当前没发生过变更的 INode 信息, 进行对应恢复。

快照之间的比差异功能对于使用者来说是非常实用的功能。因为通过比较不同时间点创建的快照, 我们可以知道在此期间到底哪些文件目录被修改、创建或删除, 甚至还能通过这些差异数据做元数据同步。快照“比差异”的命令如下:

```
hdfs snapshotDiff <path> <fromSnapshot> <toSnapshot>
```

在 HDFS 的快照中, 主要有以下 4 种变更类型:

+: 新创建的文件 / 目录

-: 被删除的文件 / 目录

M: 被修改过的文件 / 目录

R: 被重命名过的文件 / 目录

举出 HDFS 快照官方介绍的一个例子, 如果我们重命名一个目录“/foo”到“/foo2”, 并且追加新的数据到文件“/foo2/bar”, 则调用命令比较出的 diff 结果将会是如下结果:

R. /foo -> /foo2

M. /foo/bar

现在我们直接进入 snapshotDiff 命令对应的 RPC 处理代码:

```
// 快照差异比较方法
public SnapshotDiffReport diff(final INodesInPath iip,
    final String snapshotRootPath, final String from,
    final String to) throws IOException {
    // 获得当前快照的根目录
    final INodeDirectory snapshotRoot = getSnapshottableRoot(iip);

    // 如果源快照或目标快照为空, 则直接构造出 SnapshotDiffReport 差异快照信息对象
    if ((from == null || from.isEmpty())
        && (to == null || to.isEmpty())) {
        // 此情况表明源快照、目标快照与当前目录树结构一致
        return new SnapshotDiffReport(snapshotRootPath, from, to,
            Collections.<DiffReportEntry> emptyList());
    }

    // 否则先生成 SnapshotDiff 信息对象
    final SnapshotDiffInfo diffs = snapshotRoot
        .getDirectorySnapshottableFeature().computeDiff(snapshotRoot, from, to);
    // 根据 diff 信息产生报告信息对象
    return diffs != null ? diffs.generateReport() : new SnapshotDiffReport(
        snapshotRootPath, from, to, Collections.<DiffReportEntry> emptyList());
}
```

以上方法的处理又可以分为两个过程:

1) 生成 SnapshotDiffInfo 对象, 此对象里面包含了源、目标快照间发生变更的文件目

录信息。

2) 根据发生变更的文件目录信息生成 diff 报告, 展现出的形式如上述例子中所示。

(1) SnapshotDiffInfo 对象的构造产生

仔细观察上面的 2 个过程, 围绕的核心对象其实是 SnapshotDiffInfo, 快照对比信息报告也是由此对象产生。进入此类, 类内部的变量定义如下:

```
// 快照 diff 信息类
class SnapshotDiffInfo {
    ...
    // 被修改的文件、目录对象图, 按名称进行排序
    private final SortedMap<INode, byte[][]> diffMap =
        new TreeMap<INode, byte[][]>(INode_COMPARATOR);
    // 被创建或删除的文件列表图
    private final Map<INodeDirectory, ChildrenDiff> dirDiffMap =
        new HashMap<INodeDirectory, ChildrenDiff>();
    // 被重命名的对象图
    private final Map<Long, RenameEntry> renameMap =
        new HashMap<Long, RenameEntry>();
    ...
}
```

从上述变量及相应的注释中可以了解到这里维护了 3 大类信息:

- 所有被修改过的文件 / 目录, 不包括创建和删除操作。
- 所有目录下的被创建和删除的子文件。
- 所有被重命名过的文件 / 目录信息。

所以第一个子过程就是如何找出具有这些关系特征的文件目录, 并加入到这些变量中, 答案在下面这行代码所执行的操作中:

```
final SnapshotDiffInfo diffs = snapshotRoot
    .getDirectorySnapshottableFeature().computeDiff(snapshotRoot, from, to);
```

最终执行的 computeDiff 方法的代码如下:

```
private void computeDiffRecursively(final INodeDirectory snapshotRoot,
    INode node, List<byte[]> parentPath, SnapshotDiffInfo diffReport) {
    final Snapshot earlierSnapshot = diffReport.isFromEarlier() ?
        diffReport.getFrom() : diffReport.getTo();
    final Snapshot laterSnapshot = diffReport.isFromEarlier() ?
        diffReport.getTo() : diffReport.getFrom();
    byte[][] relativePath = parentPath.toArray(new byte[parentPath.size()][]);
    if (node.isDirectory()) {
        final ChildrenDiff diff = new ChildrenDiff();
        INodeDirectory dir = node.asDirectory();
        DirectoryWithSnapshotFeature sf = dir.getDirectoryWithSnapshotFeature();
        if (sf != null) {
            // 判断两个快照中的指定目录是否发生变化
            boolean change = sf.computeDiffBetweenSnapshots(earlierSnapshot,
```

```

        laterSnapshot, diff, dir);
    if (change) {
        // 如果发生了改变, 就加入到 dirDiff 中
        diffReport.addDirDiff(dir, relativePath, diff);
    }
}
...

```

首先是判断目录的变化, 在 addDirDiff 中, 会更新修改列表以及目录对应的创建 / 删除子文件的列表:

```

// 新增 diff 变更对象
void addDirDiff(INodeDirectory dir, byte[][] relativePath, ChildrenDiff diff) {
    // 新增指定目录以及对应的创建 / 删除子文件列表信息
    dirDiffMap.put(dir, diff);
    // 新增修改的目录
    diffMap.put(dir, relativePath);
    ...
}

```

然后是重命名关系的判断:

```

...
ReadOnlyList<INode> children = dir.getChildrenList(earlierSnapshot
    .getId());
for (INode child : children) {
    final byte[] name = child.getLocalNameBytes();
    boolean toProcess = diff.searchIndex(ListType.DELETED, name) < 0;
    if (!toProcess && child instanceof INodeReference.WithName) {
        byte[][] renameTargetPath = findRenameTargetPath(
            snapshotRoot, (WithName) child,
            laterSnapshot == null ? Snapshot.CURRENT_STATE_ID :
                laterSnapshot.getId());
        // 如果找到重命名对象, 则进行重命名实体更新
        if (renameTargetPath != null) {
            toProcess = true;
            diffReport.setRenameTarget(child.getId(), renameTargetPath);
        }
    }
}
...

```

最后是纯文件的变更判断:

```

...
} else if (node.isFile() && node.asFile().isWithSnapshot()) {
    INodeFile file = node.asFile();
    // 判断快照中的指定 INode 文件是否发生变化
    boolean change = file.getFileWithSnapshotFeature()
        .changedBetweenSnapshots(file, earlierSnapshot, laterSnapshot);
    // 如果发生了变化则加入到文件变更的对象变量中
    if (change) {

```

```

        diffReport.addFileDiff(file, relativePath);
    }
}

```

在 addFileDiff 中也会进行相关存储对象的更新:

```

// 新增修改对象
void addFileDiff(INodeFile file, byte[][] relativePath) {
    diffMap.put(file, relativePath);
}

```

需要注意的一点是在上述过程中会涉及自身方法的递归调用。这里有一种情况比较特殊,不同快照之间是如何判断同名的目录或文件发生了变更呢,这里举文件变更判断为例子。

```

boolean changedBetweenSnapshots(INodeFile file, Snapshot from, Snapshot to) {
    // 首先根据快照从此文件所属的 diff 列表中取出对应快照的 FileDiff 下标
    int[] diffIndexPair = diffs.changedBetweenSnapshots(from, to);
    if (diffIndexPair == null) {
        return false;
    }
    int earlierDiffIndex = diffIndexPair[0];
    int laterDiffIndex = diffIndexPair[1];

    final List<FileDiff> diffList = diffs.asList();
    // 根据对应快照的 FileDiff 下标取出当时快照下的文件大小, 进行文件大小的判断
    final long earlierLength = diffList.get(earlierDiffIndex).getFileSize();
    final long laterLength = laterDiffIndex == diffList.size() ? file
        .computeFileSize(true, false) : diffList.get(laterDiffIndex)
        .getFileSize();
    // 如果前后文件大小不一致, 则文件发生了变更
    if (earlierLength != laterLength) {
        return true;
    }
    ...
}

```

这里的意思可能有些人不太理解,之前提到过 HDFS 只是让每个快照“存不同”,然后以快照 Id 做区分。也就是说,对于同一目录,会有多个 dirDiff (dirDiff 指的是相对此快照发生改变的 INode),这些 dirDiff 被加入到了 DirDiffList 列表对象中,然后根据快照 Id 作为下标索引进行获取。同样的在文件中,也存在单个快照的 FileDiff 对象以及 FileDiffList 列表项。

整个过程如图 2-10 所示。

(2) SnapshotDiffInfo 的报告生成

上个过程结束之后,SnapshotDiffInfo 就基本构造完成了,下面是 generateReport 的过程了。这个对象的输出结果就是前面快照 diff 命令例子中所输出的信息。进入 SnapshotDiffInfo 的 generateReport 方法:

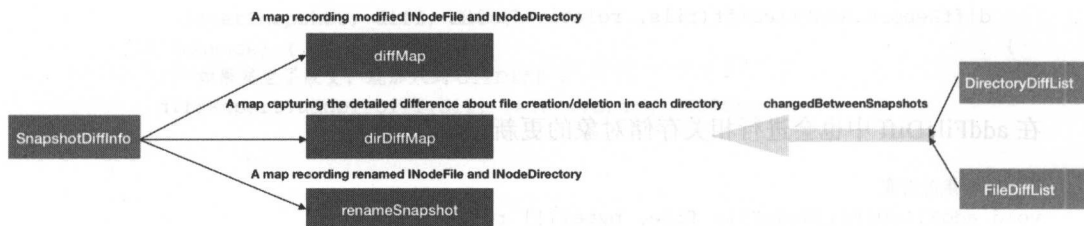


图 2-10 Snapshot 的 diff 比较生成

```

public SnapshotDiffReport generateReport() {
    List<DiffReportEntry> diffReportList = new ArrayList<DiffReportEntry>();
    // 遍历修改列表对象
    for (Map.Entry<INode,byte[][]> drEntry : diffMap.entrySet()) {
        INode node = drEntry.getKey();
        byte[][] path = drEntry.getValue();
        // 在 diffReportList 中新增 MODIFY 记录
        diffReportList.add(new DiffReportEntry(DiffType.MODIFY, path, null));
        if (node.isDirectory()) {
            // 如果是目录，则进行递归生成报告处理
            List<DiffReportEntry> subList = generateReport(dirDiffMap.get(node),
                path, isFromEarlier(), renameMap);
            // 将结果加入到 diffReportList 列表中
            diffReportList.addAll(subList);
        }
    }
    // 返回 SnapshotDiffReport 对象
    return new SnapshotDiffReport(snapshotRoot.getFullPathName(),
        Snapshot.getSnapshotName(from), Snapshot.getSnapshotName(to),
        diffReportList);
}

```

从上面的方法可以看出，主要逻辑是从修改列表的遍历开始，如果处理的 INode 是目录，则继续递归处理。生成报告的方法如下：

```

private List<DiffReportEntry> generateReport(ChildrenDiff dirDiff,
    byte[][] parentPath, boolean fromEarlier, Map<Long, RenameEntry> renameMap) {
    List<DiffReportEntry> list = new ArrayList<DiffReportEntry>();
    List<INode> created = dirDiff.getList(ListType.CREATED);
    List<INode> deleted = dirDiff.getList(ListType.DELETED);
    byte[][] fullPath = new byte[parentPath.length + 1][];
    System.arraycopy(parentPath, 0, fullPath, 0, parentPath.length);
    for (INode cnode : created) {
        RenameEntry entry = renameMap.get(cnode.getId());
        // 如果此实体不在 renameMap 中
        if (entry == null || !entry.isRename()) {
            fullPath[fullPath.length - 1] = cnode.getLocalNameBytes();
            // 判断比较的顺序，如果是晚的快照与早的快照相比，
            // 则 created 列表中的 INode 对象都是 CRATE 类型，否则相反

```

```

        list.add(new DiffReportEntry(fromEarlier ? DiffType.CREATE
                                     : DiffType.DELETE, fullPath));
    }
}
for (INode dnode : deleted) {
    RenameEntry entry = renameMap.get(dnode.getId());
    if (entry != null && entry.isRename()) {
        // 如果是重名对象, 则加入 RENAME 类型的信息记录
        list.add(new DiffReportEntry(DiffType.RENAME,
                                     fromEarlier ? entry.getSourcePath() : entry.getTargetPath(),
                                     fromEarlier ? entry.getTargetPath() : entry.getSourcePath()));
    } else {
        fullPath[fullPath.length - 1] = dnode.getLocalNameBytes();
        // 同样进行上述逻辑判断, 如果是前后快照比较的话, deleted 的 INode 都是 DELETE 类型的
        list.add(new DiffReportEntry(fromEarlier ? DiffType.DELETE
                                     : DiffType.CREATE, fullPath));
    }
}
return list;
}

```

在这里, 就会有其他 3 种类型的记录的添加。在比较的时候, 还需要注意先后快照的比较顺序, 对于不同顺序的比较, 所导致的 DiffType 是相反的。以上 4 种情况所代表的标签符号如下:

```

public enum DiffType {
    CREATE("+"),
    MODIFY("M"),
    DELETE("-"),
    RENAME("R");
    ...
}

```

SnapshotDiffReport 对象构造完毕之后, 此对象的 toString 方法输出就是命令返回的结果。总的来说, generateReport 的过程相对还是比较简单的。

2.3.4 HDFS 的快照使用

HDFS 的快照有两大用处, 下面是具体的使用场景:

- ❑ 丢失数据的恢复。这里丢失数据指的是相对于创建快照时间点之后丢失的数据。在 HDFS 的快照中, 只会额外复制发生变更的数据, 所以在快照内部, 自然会存在丢失数据的一个备份, 这个时候只需要将对应快照文件目录拷贝一份即可。
- ❑ 元数据的差异比较。HDFS 的快照能够提供 diff 比较的功能。比较的结果会展示相对于源端快照, 目标快照中发生的文件目录的变更记录。这个差异结果可以用于数据的同步, 比如快照在 DistCp 命令中的使用。用 DistCp 中的 -diff 参数附加两个 from、to 快照, 进行元数据变更的同步, 然后利用 DistCp 的功能, 进行真实数据的

拷贝，以此实现集群数据间的同步。这也是 HDFS 快照的一个很好的使用场景。当然，HDFS 快照还可以有其他使用场景，关键看你怎么去利用这个功能特性。

2.4 HDFS 副本放置策略

一个文件块从最初的产生到最后的落盘，会经过存储类型策略的选择，在存储类型选择策略中 HDFS 会帮我们先筛选一批符合存储类型要求的存储位置列表，通过这些候选列表，我们还需要做进一步的筛选，这就是本节所要讲述的主题：HDFS 的副本放置策略。HDFS 的副本放置策略主要做的事情在于副本的最终存放，位置放得好了，能提高读写性能，否则反而会起到负面的效果。我们平常所说的三副本备份策略就是其中一个副本放置策略。在本节中，我们会先讲述 HDFS 放置策略的概念，然后是 HDFS 中现有的一些放置策略，最后我们会以三副本放置策略为例，来分析放置策略的实现原理。

2.4.1 副本放置策略概念与方法

首先要先介绍什么是副本放置策略，有些文章中也会叫它副本选择策略，这源于此策略的名称：BlockPlacementPolicy。这个策略类重在副本放置（block placement），其注释说明为：“选择期望的目标节点供副本块存放。”

上面的注释已经很清楚地表达了此类型策略类的用途，接下来，我们来看下现有的一些放置策略类。

目前在 HDFS 中现有的副本放置策略类有两大继承子类，分别为 BlockPlacementPolicyDefault 和 BlockPlacementPolicyWithNodeGroup，继承关系如图 2-11 所示。

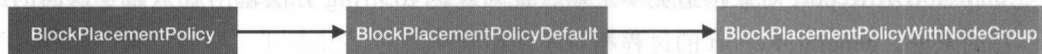


图 2-11 BlockPlacementPolicy 继承类

我们平常提到的三副本策略用的是 BlockPlacementPolicyDefault 策略类。三副本如何存放在这个策略类中得到了完美的实现。BlockPlacementPolicyDefault 类中的注释详细地解释了三个副本的存放位置，简要概括起来为以下 3 点：

1) 如果写请求方所在机器是其中一个 DataNode，则直接存放在本地，否则随机在集群中选择一个 DataNode。

2) 第二个副本存放于不同于第一个副本所在的机架。

3) 第三个副本存放于第二个副本所在的机架，但是属于不同的节点。

总的存放效果如图 2-12 所示。

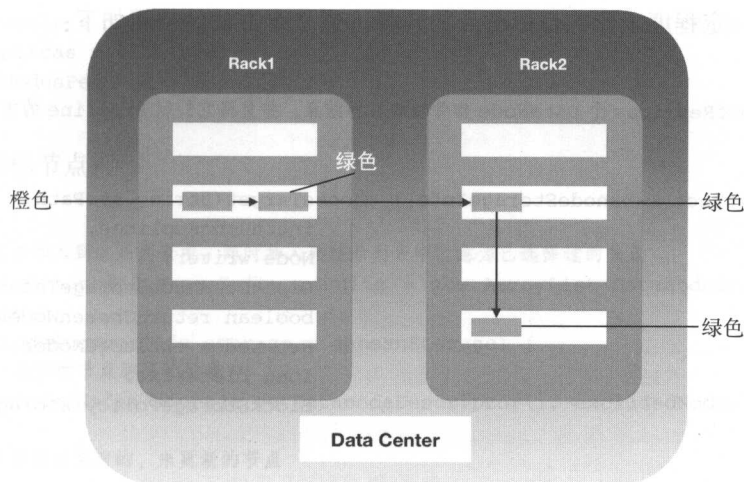


图 2-12 三副本策略放置位置

图 2-12 中橙色区域表示的是写请求者，绿色区域表示一个副本。从这里可以看出，HDFS 在容错性上还是做了很多用心的设计的。

2.4.2 副本放置策略的有效前提

如果机架感知功能关闭并不会导致副本放置策略的失败，但是副本放置策略在这种情况下会失效。因为在 HDFS 的副本放置策略中，会进行同机架、不同机架的判断。如果集群未开启机架感知功能，则默认是同一个机架，就是 `default-rack`，这会影响到 `BlockPlacementPolicy` 的逻辑处理。所以在这里需要开启这项功能，开启的方式也很简单，配置如下属性，属性值为能得到节点机架关系的脚本所在路径。

```
<property>
  <name>net.topology.script.file.name</name>
  <value>/path/to/rackAware.py</value>
</property>
```

2.4.3 默认副本放置策略的分析

`BlockPlacementPolicyDefault` 策略类中选择目标节点的处理逻辑还是有些复杂的，下面的内容会尽量简单易懂，如有不理解之处，读者可以自己对照源码进一步学习。

1. 策略核心方法 `chooseTargets`

在默认放置策略类方法中，核心方法是 `chooseTargets`。但是在这里有 2 个同名实现方法，唯一的区别是有无 `favoredNodes` 参数。`favoredNodes` 的意思是偏爱、喜爱的节点，所

以此参数会在一定程度上影响目标节点的选择。这 2 个方法的声明如下：

```
/**
 * 选择 numOfReplicas 个 DataNode 作为块的目标结点，并且将它们以 Pipeline 的方式排序返回
 *
 */
public abstract DatanodeStorageInfo[] chooseTarget(String srcPath,
                                                    int numOfReplicas,
                                                    Node writer,
                                                    List<DatanodeStorageInfo> chosen,
                                                    boolean returnChosenNodes,
                                                    Set<Node> excludedNodes,
                                                    long blocksize,
                                                    BlockStoragePolicy storagePolicy);

/**
 * 与上一方法 chooseTarget(String, int, Node, Set, long, List, StorageType) 基本类似
 * , 新添加了参数 favoredDatanodes, 表明选择目标的时候优先选择这些节点
 */
DatanodeStorageInfo[] chooseTarget(String src,
                                     int numOfReplicas, Node writer,
                                     Set<Node> excludedNodes,
                                     long blocksize,
                                     List<DatanodeDescriptor> favoredNodes,
                                     BlockStoragePolicy storagePolicy);
```

在 chooseTargets 传入偏爱的节点参数会使得方法在选择节点时候优先选取偏爱节点参数中的节点，这是此参数的最根本的影响。

然后是 chooseTarget 无 favoredNodes 参数的实现过程，此方法最终会进入到真正的同名实现方法中。下面将此过程分为了 3 个子阶段。

1) 初始化操作：

```
private DatanodeStorageInfo[] chooseTarget(int numOfReplicas,
                                           Node writer,
                                           List<DatanodeStorageInfo> chosenStorage,
                                           boolean returnChosenNodes,
                                           Set<Node> excludedNodes,
                                           long blocksize,
                                           final BlockStoragePolicy storagePolicy) {
    // 如果需要的副本数为 0 或机器节点数量为 0，返回空
    if (numOfReplicas == 0 || clusterMap.getNumOfLeaves() == 0) {
        return DatanodeStorageInfo.EMPTY_ARRAY;
    }
    // 创建移除名单列表
    if (excludedNodes == null) {
        excludedNodes = new HashSet<Node>();
    }
    // 计算每个机架所允许的最大副本数
```



```

int[] result = getMaxNodesPerRack(chosenStorage.size(), numOfReplicas);
numOfReplicas = result[0];
int maxNodesPerRack = result[1];
...

```

2) 选择目标节点:

```

...
// 将所选节点加入到结果列表中, 同时加入到移除列表中, 意为已选择过的节点
final List<DatanodeStorageInfo> results = new ArrayList<DatanodeStorageInfo>(cho
    senStorage);
for (DatanodeStorageInfo storage : chosenStorage) {
    // 添加已选择的节点到移除列表中
    addToExcludedNodes(storage.getDatanodeDescriptor(), excludedNodes);
}
// 判断是否需要避免旧的、未更新的节点
boolean avoidStaleNodes = (stats != null
    && stats.isAvoidingStaleDataNodesForWrite());
// 选择 numOfReplicas 副本数的目标机器, 并返回其中第一个节点
final Node localNode = chooseTarget(numOfReplicas, writer, excludedNodes,
    blocksize, maxNodesPerRack, results, avoidStaleNodes, storagePolicy,
// 如果不想返回初始选中的目标节点, 则进行移除
if (!returnChosenNodes) {
    results.removeAll(chosenStorage);
}
...

```

3) 对目标节点列表进行排序, 形成 Pipeline:

```

...
// 根据最短距离对目标节点列表进行排序, 形成 Pipeline
return getPipeline(
    (writer != null && writer instanceof DatanodeDescriptor) ? writer
        : localNode,
    results.toArray(new DatanodeStorageInfo[results.size()]));

```

在上述的三个子阶段中, 第二阶段是最主要的策略选择操作, 同样也是最具复杂性的, 所以这里先分析较为简单的第三个阶段的过程。第三个阶段的过程为对已经选择好的目标节点存放位置进行排序, 然后形成 Pipeline 进行返回。

Pipeline 形成的过程是传入目标节点列表参数, 经过 getPipeline 方法的处理, 然后返回此 Pipeline。简要地说, 就是从 writer 所在节点开始, 总是寻找相对路径最短的目标节点, 最终形成 Pipeline。学习过算法的同学应该知道, 这其实就是经典的 TSP 旅行商问题。下面是具体的代码实现:

```

private DatanodeStorageInfo[] getPipeline(Node writer,
    DatanodeStorageInfo[] storages) {
    if (storages.length == 0) {
        return storages;
    }
}

```

```

    }

    synchronized(clusterMap) {
        int index=0;
        // 首先如果 writer 请求方不是来自于集群中的某一个 DataNode, 则默认选取第一个 DataNode 作为起始节点
        if (writer == null || !clusterMap.contains(writer)) {
            writer = storages[0].getDatanodeDescriptor();
        }
        for(; index < storages.length; index++) {
            // 获取当前 index 下标所属的 Storage 作为最近距离的目标 Storage
            DatanodeStorageInfo shortestStorage = storages[index];
            // 计算当前距离
            int shortestDistance = clusterMap.getDistance(writer,
                shortestStorage.getDatanodeDescriptor());
            int shortestIndex = index;
            for(int i = index + 1; i < storages.length; i++) {
                // 遍历计算当前节点与剩余节点的距离
                int currentDistance = clusterMap.getDistance(writer,
                    storages[i].getDatanodeDescriptor());
                if (shortestDistance > currentDistance) {
                    shortestDistance = currentDistance;
                    shortestStorage = storages[i];
                    shortestIndex = i;
                }
            }
            // 找到新的最短距离的 Storage, 并进行下标替换
            if (index != shortestIndex) {
                storages[shortestIndex] = storages[index];
                storages[index] = shortestStorage;
            }
            // 找到当前这一轮的最近的 Storage, 并作为下一轮迭代的源节点
            writer = shortestStorage.getDatanodeDescriptor();
        }
    }
    return storages;
}

```

概括来说, 就是选出一个源节点, 根据这个节点, 遍历当前可选的下一个目标节点, 找出一个最短距离的节点, 作为下一轮选举的源节点。这样每两个节点之间的距离总是最近的, 于是整个 Pipeline 节点间的距离和就能保证是足够小的了。那么现在另外一个问题还没有解决: 如何定义和计算两个节点之间的距离。此距离的获取代码如下:

```

clusterMap.getDistance(writer,
    shortestStorage.getDatanodeDescriptor());

```

要计算其中的距离, 我们首先要了解 HDFS 中是如何定义节点间的距离的, 其中涉及拓扑逻辑的概念, 如图 2-13 所示。

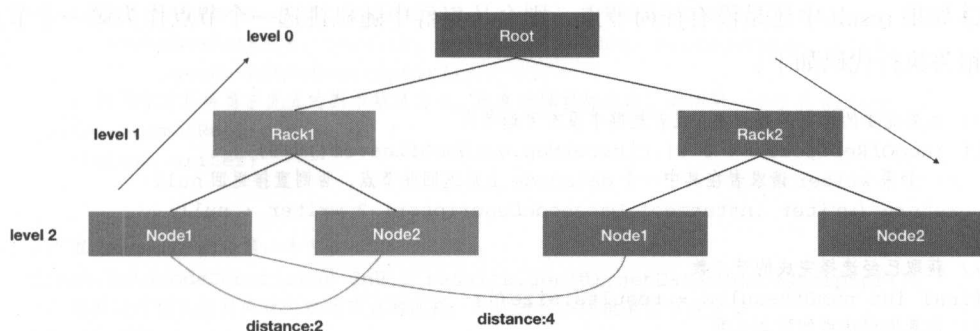


图 2-13 节点间距离定义

这里显示的是一个三层结构的树形效果图。Root 可以看做是一个大的集群，下面划分出了许多个机架，每个机架下面又有很多属于此机架的节点。在每个连接点中，是通过交换机和路由器进行连接的。每个节点间的距离计算方式是通过寻找最近公共祖先所需要的距离作为最终的结果。比如 Node1 到 Node2 的距离是 2，就是 Node1->Rack1, Rack1->Node2。同理 Rack1 的 Node1 到 Rack2 的 Node1 的距离就是 4。大家有兴趣的可以学习一下相关算法 LCA（最近公共祖先算法）。

2. chooseTarget 方法主逻辑

下面介绍 chooseTarget 主要选择逻辑。首先，务必要明确以下几个相关参数的作用和所代表的含义：

```
final Node localNode = chooseTarget(numOfReplicas, writer, excludedNodes,
    blocksize, maxNodesPerRack, results, avoidStaleNodes, storagePolicy,
    EnumSet.noneOf(StorageType.class), results.isEmpty());
```

❑ numOfReplicas: 额外需要复制的副本数。

❑ excludedNodes: 移除节点集合，此集合内的节点不应被考虑作为目标节点。

❑ results: 当前已经选择好的目标节点集合。

❑ storagePolicy: 存储类型选择策略。

(1) 首节点的选择

我们可以对照上文提到的三副本的存放方式。首先是第一个节点的选择，第一个节点其实是最好选择的，因为它不用考虑其他两个节点的位置，但是它要约束于请求方所在的位置，这里需要满足 2 个要求：

❑ 如果 writer 请求方本身位于集群中的一个 DataNode 之上，则第一个副本的位置就在本地节点上，很好理解，这样直接就是本地写操作了。如果 writer 请求方来源于外界客户端的写请求时，则从 result 列表中挑选第一个节点作为首个存放节点。

□ 如果 result 中还是没有任何节点，则会从集群中随机挑选一个节点作为第一个节点。
相关执行代码如下：

```
// 如果需要的求副本数为 0，或者集群中没有可选节点
if (numOfReplicas == 0 || clusterMap.getNumOfLeaves()==0) {
    // 如果 writer 请求者在其中一个 datanode 上则返回此节点，否则直接返回 null
    return (writer instanceof DatanodeDescriptor) ? writer : null;
}
// 获取已经选择完成的节点数
final int numOfResults = results.size();
// 计算期望达到的副本总数
final int totalReplicasExpected = numOfReplicas + numOfResults;
// 如果 writer 为空或不在 DataNode 上，则取出已选择列表中的第一个位置所在节点，赋值给 writer
if ((writer == null || !(writer instanceof DatanodeDescriptor)) && !newBlock) {
    writer = results.get(0).getDatanodeDescriptor();
}

// 做一份移除列表名单的拷贝
final Set<Node> oldExcludedNodes = new HashSet<Node>(excludedNodes);

// 根据存储策略获取副本需要满足的存储类型列表，如果有不可用的存储类型，会采用 fallback 情况下的
// storageType 类型
final List<StorageType> requiredStorageTypes = storagePolicy
    .chooseStorageTypes((short) totalReplicasExpected,
        DatanodeStorageInfo.toStorageTypes(results),
        unavailableStorages, newBlock);
// 将存储类型列表进行计数统计，并存于 map 中
final EnumMap<StorageType, Integer> storageTypes =
    getRequiredStorageTypes(requiredStorageTypes);
if (LOG.isTraceEnabled()) {
    LOG.trace("storageTypes=" + storageTypes);
}
...

```

(2) 三副本位置的选取

下面是三副本存储位置的选取过程，需要与图 2-12 展示的存放方式进行对照，会好理解一些。

```
...
// 如果 numOfReplicas 或 requiredStorageTypes 大小为 0，则抛出异常
try {
    if ((numOfReplicas = requiredStorageTypes.size()) == 0) {
        throw new NotEnoughReplicasException(
            "All required storage types are unavailable: "
            + " unavailableStorages=" + unavailableStorages
            + ", storagePolicy=" + storagePolicy);
    }
    // 如果已选择的目标节点数量为 0，则表示三副本一个都还没开始选，首先从选本地节点开始
    if (numOfResults == 0) {

```

```

writer = chooseLocalStorage(writer, excludedNodes, blocksize,
    maxNodesPerRack, results, avoidStaleNodes, storageTypes, true)
    .getDataNodeDescriptor();
// 如果此时目标需求完成的副本数降为 0, 代表选择目标完成, 返回第一个节点 writer
if (--numOfReplicas == 0) {
    return writer;
}
}
// 取出 result 列表第一个节点
final DatanodeDescriptor dn0 = results.get(0).getDataNodeDescriptor();
// 前面的过程已经完成首个本地节点的选择, 此时进行不同机架的节点选择
if (numOfResults <= 1) {
    // 选择 1 个不同于 dn0 所在机房的一个目标节点位置
    chooseRemoteRack(1, dn0, excludedNodes, blocksize, maxNodesPerRack,
        results, avoidStaleNodes, storageTypes);
    // 如果此时目标需求完成的副本数降为 0, 代表选择目标完成, 返回第一个节点 writer
    if (--numOfReplicas == 0) {
        return writer;
    }
}
// 如果经过前面的处理, 节点选择数在 2 个以内, 需要选取第三个副本
if (numOfResults <= 2) {
    // 取出 result 列表第二个节点
    final DatanodeDescriptor dn1 = results.get(1).getDataNodeDescriptor();
    // 如果 dn0、dn1 在同机房
    if (clusterMap.isOnSameRack(dn0, dn1)) {
        // 则选择 1 个不同于 dn0, dn1 所在机房的副本位置
        chooseRemoteRack(1, dn0, excludedNodes, blocksize, maxNodesPerRack,
            results, avoidStaleNodes, storageTypes);
    } else if (newBlock) {
        // 如果是新的块, 则选取 1 个与 dn1 同机房的节点位置
        chooseLocalRack(dn1, excludedNodes, blocksize, maxNodesPerRack,
            results, avoidStaleNodes, storageTypes);
    } else {
        // 否则选取与 writer 同机房的位置
        chooseLocalRack(writer, excludedNodes, blocksize, maxNodesPerRack,
            results, avoidStaleNodes, storageTypes);
    }
    // 如果此时目标需求完成的副本数降为 0, 代表选择目标完成, 返回第一个节点 writer
    if (--numOfReplicas == 0) {
        return writer;
    }
}
// 如果副本数已经超过 2 个, 说明设置块时已经设置超过三副本的数量
// 则剩余位置在集群中随机选择放置节点
chooseRandom(numOfReplicas, NodeBase.ROOT, excludedNodes, blocksize,
    maxNodesPerRack, results, avoidStaleNodes, storageTypes);

```

如果看完这段逻辑, 你还不理解的话, 没有关系, 只要明白经典的三副本存放位置, 多余的副本随机存放的原理即可。当然在选择的过程中可能会发生异常, 有时我们没有配

置机架感知，集群中的节点都属于一个默认机架（default-rack），会导致 chooseRemoteRack 的方法出错，因为没有满足条件的其余机架。这时需要一些重试策略，代码如下所示：

```
if (retry) {
    for (DatanodeStorageInfo resultStorage : results) {
        addToExcludedNodes(resultStorage.getDatanodeDescriptor(),
            oldExcludedNodes);
    }
    // 剔除之前选择完成的目标位置，重新计算当前需要复制的副本数
    numOfReplicas = totalReplicasExpected - results.size();
    // 重新调用自身方法进行复制块目标节点的选择
    return chooseTarget(numOfReplicas, writer, oldExcludedNodes, blocksize,
        maxNodesPerRack, results, false, storagePolicy, unavailableStorages,
        newBlock);
}
```

（3）chooseLocalStorage、chooseLocalRack、chooseRemoteRack 和 chooseRandom 方法这 4 个选择目标节点位置的方法是一个优先级逐级降低的方法。首先选择本地存储位置，如果没有满足条件的节点，再选择本地机架的节点，如果还是没有满足条件的节点，进一步降级选择不同机架的节点，最后随机选择集群中的节点。降级选择过程见图 2-14。

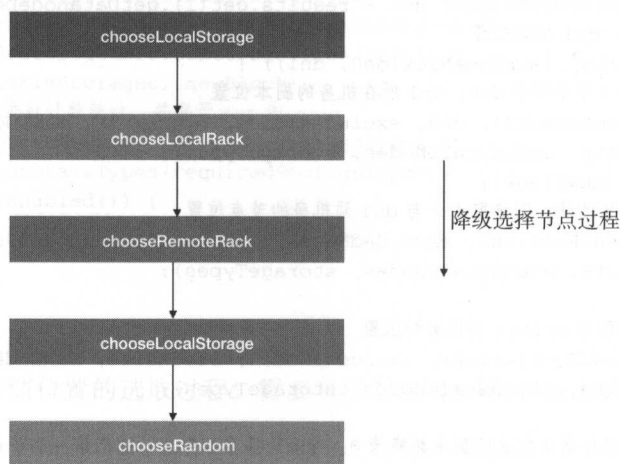


图 2-14 选择节点操作

但是这里还是要区分一下 chooseLocalStorage 方法。此方法与其余的 3 个方法稍显不同，它是单独实现的，而其余的方法是通过传入不同参数直接或间接调用 chooseRandom 方法进行构造的。

首先来看 chooseLocalStorage 方法实现：

```
protected DatanodeStorageInfo chooseLocalStorage(Node localMachine,
    Set<Node> excludedNodes, long blocksize, int maxNodesPerRack,
```

```

List<DatanodeStorageInfo> results, boolean avoidStaleNodes,
EnumMap<StorageType, Integer> storageTypes, boolean fallbackToLocalRack)
throws NotEnoughReplicasException {
// if no local machine, randomly choose one node
if (localMachine == null) {
// 如果本地节点为空, 则降级选择一个随机节点
return chooseRandom(NodeBase.ROOT, excludedNodes, blocksize,
maxNodesPerRack, results, avoidStaleNodes, storageTypes);
}
if (preferLocalNode && localMachine instanceof DatanodeDescriptor) {
DatanodeDescriptor localDatanode = (DatanodeDescriptor) localMachine;
// 否则尝试选择本地节点
if (excludedNodes.add(localMachine)) {

```

如果本地节点没有被包含在移除列表中, 则进行下面的操作:

```

for (Iterator<Map.Entry<StorageType, Integer>> iter = storageTypes
.entrySet().iterator(); iter.hasNext(); ) {
Map.Entry<StorageType, Integer> entry = iter.next();
// 遍历本地节点可用的存储目录
for (DatanodeStorageInfo localStorage : DFSUtil.shuffle(
localDatanode.getStorageInfos())) {
StorageType type = entry.getKey();
// 加入满足条件的存储目录位置
if (addIfIsGoodTarget(localStorage, excludedNodes, blocksize,
maxNodesPerRack, false, results, avoidStaleNodes, type) >= 0) {
...
// 本地节点没有满足条件的存储位置, 则降级选取同机架的节点
return chooseLocalRack(localMachine, excludedNodes, blocksize,
maxNodesPerRack, results, avoidStaleNodes, storageTypes);
}
}

```

chooseLocalRack 和 chooseRemoteRack 比较类似。

chooseLocalRack 方法如下:

```

// 没有本地节点, 则选择一个随机的节点
if (localMachine == null) {
return chooseRandom(NodeBase.ROOT, excludedNodes, blocksize,
maxNodesPerRack, results, avoidStaleNodes, storageTypes);
}
// 获取本地机架名
final String localRack = localMachine.getNetworkLocation();

try {
// 将机架名作为 scope 参数传入
return chooseRandom(localRack, excludedNodes,
blocksize, maxNodesPerRack, results, avoidStaleNodes, storageTypes);
}

```

chooseRemoteRack 的方法如下:

```
// 获取本地机架名称，带上前缀字符“~”，作为 scope 参数传入
chooseRandom(numOfReplicas, "~" + localMachine.getNetworkLocation(),
    excludedNodes, blocksize, maxReplicasPerRack, results,
    avoidStaleNodes, storageTypes);
```

从这里我们可以看到，其中最明显的区别是 `chooseRandom` 中 `scope` 参数的传入，`scope` 参数的作用是选择出属于此机架下的节点列表。

```
DatanodeDescriptor chosenNode =
    (DatanodeDescriptor)clusterMap.chooseRandom(scope);
```

在 `NetworkTopology` 下是其具体的实现：

```
/**
 * 从 scope 所示范围中随机选取一个节点，如果 scope 以字符“~”打头，则从非 scope 范围
 * 节点中选取节点
 */
public Node chooseRandom(String scope) {
    netlock.readLock().lock();
    try {
        if (scope.startsWith("~")) {
            return chooseRandom(NodeBase.ROOT, scope.substring(1));
        } else {
            return chooseRandom(scope, null);
        }
    } finally {
        netlock.readLock().unlock();
    }
}
```

具体的实现细节，读者可以自行研究。机架节点选择好之后，接着会进行 `Storage` 存储位置的选择判断，然后加入到 `result` 目标列表中。

2.4.4 目标存储好坏的判断

如果块放置节点已经初步选择好了，是否意味着此位置就可以加入最终的 `result` 列表中呢？答案是否定的，因为这里还要经过最后一道对于存储的验证（这里要明确一点：目标位置 `result` 类别存储的对象是 `DatanodeStorageInfo`，这个类表示的是具体到节点存储磁盘目录级别的信息，并不是广义上的节点），需要满足以下几个条件：

- ❑ 存储的存储类型必须是请求的存储类型。
- ❑ 存储不能是 `READ_ONLY`（只读）。
- ❑ 存储不能是坏的。
- ❑ 存储所在节点不应该是已下线或下线中的节点。
- ❑ 存储所在节点不应该是消息落后的节点，实际指的是一段时间内没有更新心跳的节点。


```

        short replication,
        long blockSize,
        Progressable progress,
        int buffersize,
        ChecksumOpt checksumOpt)

    throws IOException {
    return create(src, permission, flag, true,
        replication, blockSize, progress, buffersize, checksumOpt, null);
    }

```

最后一个 null 就是传入的 `favoredNodes` 参数。其实传入的 `favoredNodes` 更多的是一种期望，最后并不一定能被 `NameNode` 真正存放。因为中间会经过很多因素的影响，而且在后面的 `Balance`（数据平衡）的过程中，某些块还是会被挪走，就不会按照原来的位置存放。

2.4.6 BlockPlacementPolicyWithNodeGroup 继承类

`BlockPlacementPolicyWithNodeGroup` 是 `BlockPlacementPolicyDefault` 的继承子类。前者与后者在原理上十分类似，不过在逻辑上从机架是否相同的判断变为了是否为同个 `Node-Group` 的判断，详细解释可阅读其源码注释。

它是一个四层层级结构，在 `Rack` 机架层下还多了 `Node-Group` 层，结构图如图 2-17 所示。

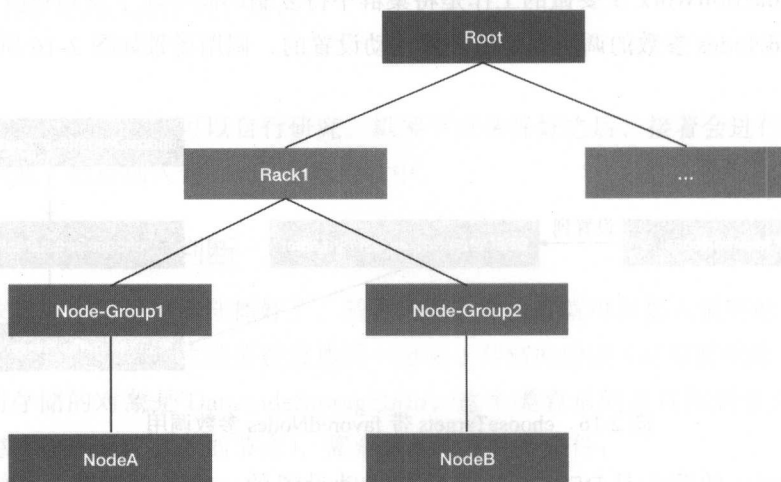


图 2-17 NodeGroup 的拓扑结构

由于与其父类的逻辑没有很大差别，这里就不展开讲述了。

2.4.7 副本放置策略的结果验证

如果一个副本块最终完成了所有的写操作并且已经完全存入到相应的 DataNode 中了,这时我们如何验证三副本存放策略的有效性呢?换句话说,我们需要有一种方式能够检测块当前的详细位置,这样我们才能判断是否满足 HDFS 的副本放置策略。这里介绍一个相关的检测命令: fsck。fsck 命令是专门用于做块检查的命令。执行如下命令可以得到文件所属块的详细位置信息。

```
hdfs fsck <path> -files -blocks -locations
```

以上内容就是本节所要讲述的 HDFS 放置策略的内容了,在内容量上可能有点多,希望大家通过此节内容的学习能对 HDFS 的三副本策略以及背后的放置策略有更深入的了解。

2.5 HDFS 内部的认证机制

数据的安全性是一直被大家所重视的。对于一个存有大规模数据的成熟企业来说,如何做到数据不丢失、不损坏、不窃取是十分重要的。如果我们用 HDFS 存储大规模的数据,如何保证其中数据的安全性呢?本节我们将介绍 HDFS 内部两大认证机制: BlockToken 认证和 Sasl 认证。这两个认证机制在一定程度上能起到数据保护的作用。BlockToken 认证是基于令牌的块级别粒度的验证,而 Sasl 认证则是标准的 Sasl 认证机制在 HDFS 中的具体实现。在本节的最后,我们会对这两大机制做一个简单的比较,让大家更加清晰地了解二者的异同。

2.5.1 BlockToken 认证

相比较而言,BlockToken 认证比 Sasl 认证要简单一些,而且 BlockToken 在 Sasl 认证中也会用到。细看 BlockToken 这个词,可以将其拆分为两个词: Block 和 Token,由此我们可以得出以下两点关键信息:

□ BlockToken 是针对块级别的认证。

□ Token 是“令牌”的意思,通常是用做访问时的认证。

大体有了这么一个了解之后,接下来看 BlockToken 机制是如何做具体认证的,要完全理解这个问题,需要弄清楚以下三点:

□ BlockToken 如何生成?

□ BlockToken 在哪里认证?

□ BlockToken 如何认证?

1. BlockToken 的结构分析

BlockToken 的结构分析可以帮助我们了解它是如何产生的。如果你仔细观察和查找,应该很容易就找到相关类:BlockPoolTokenSecretManager。BlockToken 就是由这个类调用产生的。但是真正产生 BlockToken 的操作实质上是由其存储的 BlockTokenSecretManager 类做的。所以这些类的关系为:BlockPoolTokenSecretManager 包含 BlockTokenSecretManager,并且每一个 BlockPool 对应一个 BlockTokenSecretManager。

最终是如下的存储映射关系:

```
private final Map<String, BlockTokenSecretManager> map =
    new HashMap<String, BlockTokenSecretManager>();
```

可能会有人有问,为什么按照 BlockPool 分出这么多的 BlockTokenSecretManager,全局维护一个 Manager 不是更好吗?笔者的个人看法是 HDFS 这么做还是想做隔离,BlockPool 是在每次 NameNode 做 format 时产生的,代表着独立的存储空间和命名空间。所有的块在各自所属的 BlockPool 下是全局唯一的,每个 BlockPool 下的块也是由独立所属的 BlockManager (块管理器)进行管理。HDFS 中是可以有多个 BlockPool 的。回到前面说的过程,继续来看 BlockToken 的生成调用过程。

```
// Token 生成方法
public Token<BlockTokenIdentifier> generateToken(ExtendedBlock b,
    EnumSet<AccessMode> of) throws IOException {
    // 选择块所属的 BlockPool 去生成 Token
    return get(b.getBlockPoolId()).generateToken(b, of);
}
```

实际的调用方法如下所示:

```
public Token<BlockTokenIdentifier> generateToken(ExtendedBlock block,
    EnumSet<BlockTokenIdentifier.AccessMode> modes) throws IOException {
    UserGroupInformation ugi = UserGroupInformation.getCurrentUser();
    String userID = (ugi == null ? null : ugi.getShortUserName());
    return generateToken(userID, block, modes);
}

// 生成 Token 最终调用方法
public Token<BlockTokenIdentifier> generateToken(String userID,
    ExtendedBlock block, EnumSet<BlockTokenIdentifier.AccessMode> modes) throws
    IOException {
    // 将块相关信息、用户信息、访问模式信息设置入 BlockToken 对象中,并返回
    BlockTokenIdentifier id = new BlockTokenIdentifier(userID, block
        .getBlockPoolId(), block.getBlockId(), modes);
    return new Token<BlockTokenIdentifier>(id, this);
}
```

BlockToken 在创建块的时候会被构建:

```
private LocatedBlock createLocatedBlock(final BlockInfo blk, final long pos,
    final AccessMode mode) throws IOException {
    final LocatedBlock lb = createLocatedBlock(blk, pos);
    // 设置 BlockToken
    if (mode != null) {
        setBlockToken(lb, mode);
    }
    return lb;
}
```

setBlockToken 方法的代码如下：

```
public void setBlockToken(final LocatedBlock b,
    final AccessMode mode) throws IOException {
    // 如果开启了 BlockToken 认证功能
    if (isBlockTokenEnabled()) {
        // Use cached UGI if serving RPC calls.
        if (b.isStriped()) {
            Preconditions.checkNotNull(b instanceof LocatedStripedBlock);
            LocatedStripedBlock sb = (LocatedStripedBlock) b;
            byte[] indices = sb.getBlockIndices();
            Token<BlockTokenIdentifier>[] blockTokens = new Token[indices.length];
            ExtendedBlock internalBlock = new ExtendedBlock(b.getBlock());
            for (int i = 0; i < indices.length; i++) {
                internalBlock.setBlockId(b.getBlock().getBlockId() + indices[i]);
                // 生成 BlockToken 对象
                blockTokens[i] = blockTokenSecretManager.generateToken(
                    NameNode.getRemoteUser().getShortUserName(),
                    internalBlock, EnumSet.of(mode));
            }
            sb.setBlockTokens(blockTokens);
        } else {
            // 生成 BlockToken 对象并设置到块中
            b.setBlockToken(blockTokenSecretManager.generateToken(
                NameNode.getRemoteUser().getShortUserName(),
                b.getBlock(), EnumSet.of(mode)));
        }
    }
}
```

这就是 **BlockToken** 从产生到被设置到目标对象的过程。注意上述代码中的一个细节处理，即加粗的代码。也就是说，**BlockToken** 功能是可控的，它是一个受配置控制的功能，具体的配置项后面会具体说明。综上所述，在这个部分我们基本上了解了 **BlockToken** 相关的结构设计以及相关的方法，如图 2-18 所示。

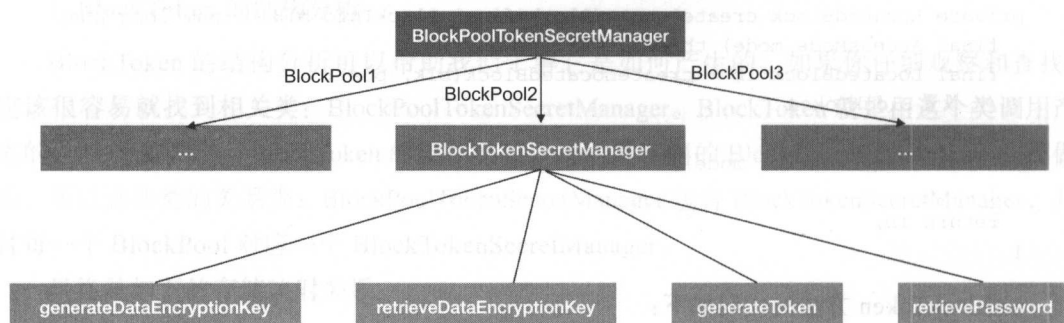


图 2-18 BlockToken 相关结构

2. BlockToken 的调用与认证

BlockToken 的认证是在 BlockTokenSecretManager 的 checkAccess 方法中执行的，接下来我们需要找到这个方法的调用处，答案在 DataXceiver 类中。DataXceiver 是一个数据处理中心，它会接收各种块操作命令，然后执行对应的处理方法。而在 readBlock、writeBlock 执行的前几步操作中，就包含了 BlockToken 的认证操作。以 readBlock 方法为例：

```

public void readBlock(final ExtendedBlock block,
    final Token<BlockTokenIdentifier> blockToken,
    final String clientName,
    final long blockOffset,
    final long length,
    final boolean sendChecksum,
    final CachingStrategy cachingStrategy) throws IOException {
    previousOpClientName = clientName;
    long read = 0;
    updateCurrentThreadName("Sending block " + block);
    OutputStream baseStream = getOutputStream();
    DataOutputStream out = getBufferedOutputStream();
    // 进行 Token READ 访问模式的验证
    checkAccess(out, true, block, blockToken,
        Op.READ_BLOCK, BlockTokenIdentifier.AccessMode.READ);
    ...
}

```

checkAccess 方法的代码如下：

```

private void checkAccess(OutputStream out, final boolean reply,
    final ExtendedBlock blk,
    final Token<BlockTokenIdentifier> t,
    final Op op,
    final BlockTokenIdentifier.AccessMode mode) throws IOException {
    checkAndWaitForBP(blk);
    // 进行是否已启用 BlockToken 验证的判断
    if (datanode.isBlockTokenEnabled) {

```

```

if (LOG.isDebugEnabled()) {
    LOG.debug("Checking block access token for block '" + blk.getBlockId()
        + "' with mode '" + mode + "'");
}
try {
    // 进行 BlockToken 的访问验证
    datanode.blockPoolTokenSecretManager.checkAccess(t, null, blk, mode);
} catch (InvalidToken e) {
    // 验证异常处理
    ...
}

```

在最终的 `checkAccess` 方法中, 进行多指标维度的信息认证:

```

public void checkAccess(Token<BlockTokenIdentifier> token, String userId,
    ExtendedBlock block, BlockTokenIdentifier.AccessMode mode) throws
    InvalidToken {
    BlockTokenIdentifier id = new BlockTokenIdentifier();
    try {
        // 反序列化 Token
        id.readFields(new DataInputStream(new ByteArrayInputStream(token
            .getIdentifier())));
    } catch (IOException e) {
        throw new InvalidToken(
            "Unable to de-serialize block token identifier for user=" + userId
            + ", block=" + block + ", access mode=" + mode);
    }
    // 进行相关信息的验证
    checkAccess(id, userId, block, mode);
    // 进行密码的验证
    if (!Arrays.equals(retrievePassword(id), token.getPassword())) {
        throw new InvalidToken("Block token with " + id.toString()
            + " doesn't have the correct token password");
    }
}

```

继续进入 `id`、`userId` 等相关信息的认证方法:

```

public void checkAccess(BlockTokenIdentifier id, String userId,
    ExtendedBlock block, BlockTokenIdentifier.AccessMode mode) throws
    InvalidToken {
    if (LOG.isDebugEnabled()) {
        LOG.debug("Checking access for user=" + userId + ", block=" + block
            + ", access mode=" + mode + " using " + id.toString());
    }
    // 用户 Id 验证
    if (userId != null && !userId.equals(id.getUserId())) {
        throw new InvalidToken("Block token with " + id.toString()
            + " doesn't belong to user " + userId);
    }
    // BlockPoolId 验证
    if (!id.getBlockPoolId().equals(block.getBlockPoolId())) {

```

```

        throw new InvalidToken("Block token with " + id.toString()
            + " doesn't apply to block " + block);
    }
    // 块 ID 验证
    if (id.getBlockId() != block.getBlockId()) {
        throw new InvalidToken("Block token with " + id.toString()
            + " doesn't apply to block " + block);
    }
    // 过期验证
    if (isExpired(id.getExpiryDate())) {
        throw new InvalidToken("Block token with " + id.toString()
            + " is expired.");
    }
    // 访问模式验证
    if (!id.getAccessModes().contains(mode)) {
        throw new InvalidToken("Block token with " + id.toString()
            + " doesn't have " + mode + " permission");
    }
}
}

```

通过上述过程的分析，可以看出 BlockToken 的认证是非常严格的，一旦 Token 中某个指标信息不匹配，马上会抛出异常，后续的方法也随之无法继续进行。HDFS 将 BlockToken 认证处理放在 DataXceiver 中，进行全局的控制，显然是精密思考的选择。图 2-19 是 BlockToken 认证的流程示意图。

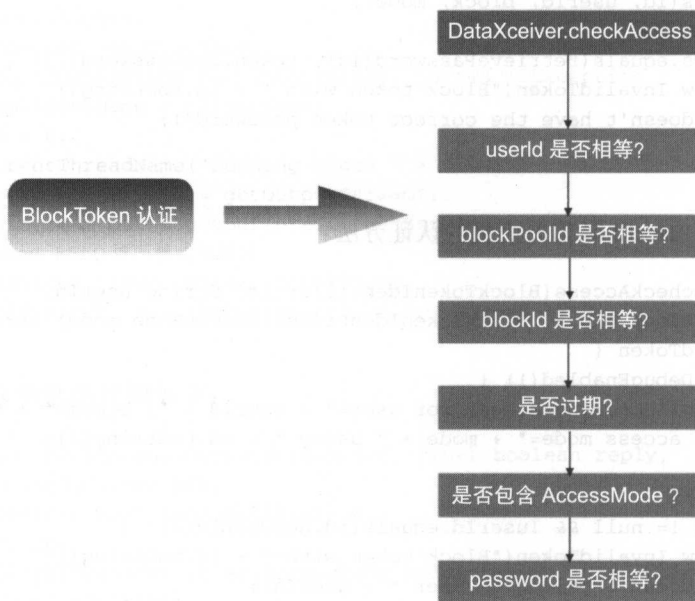


图 2-19 BlockToken 认证流程

3. BlockToken 认证配置

下面介绍 BlockToken 的配置控制，配置项名称如下：

```
dfs.block.access.token.enable
```

该配置项默认不开启，就是 false 状态。如果我们出于数据安全性的考虑，启用了块认证的功能，需要注意的一点是它可能会对 HDFS 的块读写性能造成影响。因为里面有一些反序列化操作和很多的认证操作，而且这些操作在每次的块操作中都会进行。

2.5.2 HDFS 的 Sasl 认证

这一节将关注点移向 HDFS 中的另外一套认证体系：Sasl 认证。说起 Sasl，它不是 HDFS 所特有的，它是一套公开的认证机制，全称是 Simple Authentication and Security Layer，中文翻译为“简单认证与安全层”，是一种用来扩充 C/S 模式验证能力的机制。

那么 HDFS 中的 Sasl 与平常我们所说的 Sasl 机制有什么不同呢？笔者认为没有本质的区别，只是将 Sasl 认证体系整合进了 HDFS 的数据读写流程中。

所以如果你之前了解过 Sasl，或者用过这套体系，可能后面的内容会比较容易理解。在下面的内容中，主要选出了 Sasl 与 HDFS 结合的部分进行分析。

1. SaslClient 与 SaslServer 的握手

在上面的内容中已经介绍过 Sasl 是 C/S 模式的认证机制，所以自然地会分出两个角色：SaslClient 和 SaslServer。在每次的数据传输中，客户端与服务端都会进行一次握手 (handshake)。比如在 DataStreamer 的 transfer 方法中：

```
private void transfer(final DatanodeInfo src, final DatanodeInfo[] targets,
    final StorageType[] targetStorageTypes,
    final Token<BlockTokenIdentifier> blockToken)
    throws IOException {
    // 传输副本到新的节点
    Socket sock = null;
    DataOutputStream out = null;
    DataInputStream in = null;
    try {
        sock = createSocketForPipeline(src, 2, dfsClient);
        final long writeTimeout = dfsClient.getDatanodeWriteTimeout(2);

        // 数据传输超时时间的计算，时间长短取决于传输数据包的大小
        int multi = 2 + (int)(bytesSent / dfsClient.getConf().getWritePacketSize())
            / 200;
        final long readTimeout = dfsClient.getDatanodeReadTimeout(multi);

        OutputStream unbufOut = NetUtils.getOutputStream(sock, writeTimeout);
        InputStream unbufIn = NetUtils.getInputStream(sock, readTimeout);
```

```

// SaslClient 与服务端建立一次握手
IOStreamPair saslStreams = dfsClient.saslClient.socketSend(sock,
    unbufOut, unbufIn, dfsClient, blockToken, src);
// 获取结果输入流和输出流
unbufOut = saslStreams.out;
unbufIn = saslStreams.in;
...

```

略过 `dfsClient.saslClient` 的中间处理方法，进入最终的握手处理方法：

```

private IOStreamPair send(InetAddress addr, OutputStream underlyingOut,
    InputStream underlyingIn, DataEncryptionKey encryptionKey,
    Token<BlockTokenIdentifier> accessToken, DatanodeID datanodeId)
    throws IOException {
    // 如果加密 key 不为空，则进行加密握手方式
    if (encryptionKey != null) {
        LOG.debug("SASL client doing encrypted handshake for addr = {}, "
            + "datanodeId = {}", addr, datanodeId);
        return getEncryptedStreams(addr, underlyingOut, underlyingIn,
            encryptionKey);
    } else if (!UserGroupInformation.isSecurityEnabled()) {
        // 如果安全配置没有开启，则跳过握手操作
        LOG.debug("SASL client skipping handshake in unsecured configuration for "
            + "addr = {}, datanodeId = {}", addr, datanodeId);
        return null;
    } else if (SecurityUtil.isPrivilegedPort(datanodeId.getXferPort())) {
        // 如果是特权端口号，也不做处理
        LOG.debug(
            "SASL client skipping handshake in secured configuration with "
            + "privileged port for addr = {}, datanodeId = {}",
            addr, datanodeId);
        return null;
    } else if (fallbackToSimpleAuth != null && fallbackToSimpleAuth.get()) {
        // 如果是简单认证模式，也跳过握手操作
        LOG.debug(
            "SASL client skipping handshake in secured configuration with "
            + "unsecured cluster for addr = {}, datanodeId = {}",
            addr, datanodeId);
        return null;
    } else if (saslPropsResolver != null) {
        // 进行普通方式的握手
        LOG.debug(
            "SASL client doing general handshake for addr = {}, datanodeId = {}",
            addr, datanodeId);
        return getSaslStreams(addr, underlyingOut, underlyingIn, accessToken);
    } else {
        // 如果是其他情况，则忽略处理，返回 null
        LOG.debug("SASL client skipping handshake in secured configuration with "
            + "no SASL protection configured for addr = {}, datanodeId = {}",
            addr, datanodeId);
    }
}

```

```

    return null;
}
}

```

这里会进行多重因素的判断，以此决定用何种握手方式。大体分为以下几种：

- ❑ 加密 key 不为空，则进行加密握手操作。
- ❑ 未开启安全配置模式，不进行握手操作。
- ❑ Sasl 相关配置项不为空，进行普通握手操作。
- ❑ 如果是特权端口号，不进行握手操作。
- ❑ 如果是简单认证模式，不进行握手操作。
- ❑ 其他情况，同样不进行握手操作。

所以真正做握手操作的只有两种情况，如图 2-20 所示。

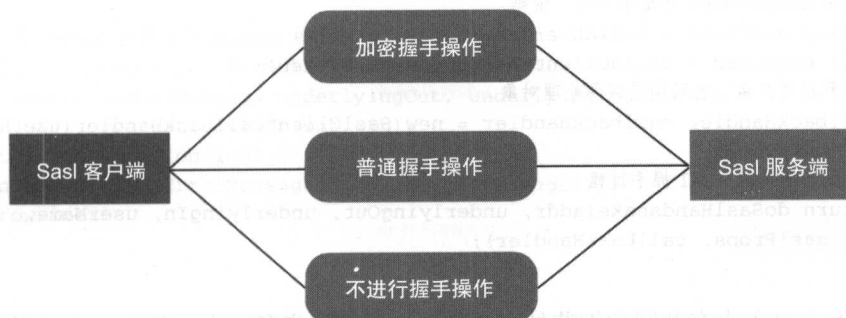


图 2-20 HDFS 的 Sasl 握手方式

如果我们什么安全配置都没有开启的话，也就是沿用默认值的情况，握手逻辑将从握手处理方法的第二个 if 判断逻辑中跳过。

2. DoSaslHandshake

在这两类进入真正握手阶段的方法中，会提前一步进行用户、密码的构造过程。

首先是加密握手阶段的构造过程：

```

private IOStreamPair getEncryptedStreams(InetAddress addr,
    OutputStream underlyingOut,
    InputStream underlyingIn, DataEncryptionKey encryptionKey)
    throws IOException {
    Map<String, String> saslProps = createSaslPropertiesForEncryption(
        encryptionKey.encryptionAlgorithm);

    LOG.debug("Client using encryption algorithm {}",
        encryptionKey.encryptionAlgorithm);
    // 用 encryptionKey 构造用户名、密码
}

```

```

String userName = getUserNameFromEncryptionKey(encryptionKey);
char[] password = encryptionKeyToPassword(encryptionKey.encryptionKey);
// 利用用户名、密码构造回调处理对象
CallbackHandler callbackHandler = new SaslClientCallbackHandler(userName,
    password);
// 执行具体的 sasl 握手过程
return doSaslHandshake(addr, underlyingOut, underlyingIn, userName,
    saslProps, callbackHandler);
}

```

第二个是普通方式握手的构造过程：

```

private IOStreamPair getSaslStreams(InetAddress addr,
    OutputStream underlyingOut, InputStream underlyingIn,
    Token<BlockTokenIdentifier> accessToken)
    throws IOException {
    Map<String, String> saslProps = saslPropsResolver.getClientProperties(addr);
    // 用 BlockToken 构造用户名、密码
    String userName = buildUserName(accessToken);
    char[] password = buildClientPassword(accessToken);
    // 利用用户名、密码构造回调处理对象
    CallbackHandler callbackHandler = new SaslClientCallbackHandler(userName,
        password);
    // 执行具体的 sasl 握手过程
    return doSaslHandshake(addr, underlyingOut, underlyingIn, userName,
        saslProps, callbackHandler);
}

```

上述两个过程中有共同的细节处理操作：生成的用户名、密码都被 Base64 编码处理过。比如其中的一个例子：

```

public static char[] encryptionKeyToPassword(byte[] encryptionKey) {
    return new String(Base64.encodeBase64(encryptionKey, false), Charsets.UTF_8)
        .toCharArray();
}

```

Base64 编码的处理可以防止明文的暴露。经过以上处理之后，最终会执行真正的握手阶段，也就是 Sasl 的认证阶段，客户端与服务端将会进行身份信息的交换认证。

首先由 SaslClient 端发起，执行代码如下：

```

private IOStreamPair doSaslHandshake(InetAddress addr,
    OutputStream underlyingOut, InputStream underlyingIn, String userName,
    Map<String, String> saslProps,
    CallbackHandler callbackHandler) throws IOException {
    ...
    try {
        // 初始握手阶段。
        sendSaslMessage(out, new byte[0]);
        // 下面进行客户端询问、回复操作
    }
}

```

```

// step 1
byte[] remoteResponse = readSaslMessage(in);
byte[] localResponse = sasl.evaluateChallengeOrResponse(remoteResponse);
...
sendSaslMessageAndNegotiationCipherOptions(out, localResponse,
    cipherOptions);

// Step 2 (client-side only)
SaslResponseWithNegotiatedCipherOption response =
    readSaslMessageAndNegotiatedCipherOption(in);
localResponse = sasl.evaluateChallengeOrResponse(response.payload);
assert localResponse == null;

// Sasl 握手完成
checkSaslComplete(sasl, saslProps);

...

// 用 cipher 参数构造 stream 数据流
return cipherOption != null ? createStreamPair(
    conf, cipherOption, underlyingOut, underlyingIn, false) :
    sasl.createStreamPair(out, in);
} catch (IOException ioe) {
    sendGenericSaslErrorMessage(out, ioe.getMessage());
    throw ioe;
}
}
}

```

然后是对应的 SaslServer 处理回应操作:

```

private IOStreamPair doSaslHandshake(Peer peer, OutputStream underlyingOut,
    InputStream underlyingIn, Map<String, String> saslProps,
    CallbackHandler callbackHandler) throws IOException {
    ...
    try {
        // 下面进行服务端的询问、回复操作
        // step 1
        byte[] remoteResponse = readSaslMessage(in);
        byte[] localResponse = sasl.evaluateChallengeOrResponse(remoteResponse);
        sendSaslMessage(out, localResponse);

        // step 2 (server-side only)
        List<CipherOption> cipherOptions = Lists.newArrayList();
        // 读取 sasl 信息并生成回复
        remoteResponse = readSaslMessageAndNegotiationCipherOptions(
            in, cipherOptions);
        localResponse = sasl.evaluateChallengeOrResponse(remoteResponse);

        // Sasl 握手完成
        checkSaslComplete(sasl, saslProps);
    }
}

```

```

...
// 用 cipher 参数构造 stream 数据流
return cipherOption != null ? createStreamPair(
    dnConf.getConf(), cipherOption, underlyingOut, underlyingIn, true) :
    sasl.createStreamPair(out, in);
} catch (IOException ioe) {
    ...
}
}

```

在握手操作完成之后，将会得到一个新的输入输出流对象。如果我们没有配置 cipher option 加密参数，将会默认使用 SaslInputStream 和 SaslOutputStream。此部分的 Sasl 机制执行逻辑与平常我们所说的 Sasl 基本一致。图 2-21 是对应的流程图。



图 2-21 HDFS 的 Sasl 握手过程

大致总结一下图 2-21 中 doSaslHandshake 所示的过程：

- 1) 首先客户端发送初始请求。
- 2) 服务端收到请求，生成询问，发送给客户端。
- 3) 客户端收到询问，处理询问，生成回复，发送给服务端。
- 4) 服务端收到询问回复，验证回复，验证通过后，再给客户端一个响应回复，进行确认。
- 5) 双方都确认完毕，握手结束。

当然，在握手阶段如果发生了失败或异常，同样会导致后续操作的失败。

3. SaslInputStream 和 SaslOutputStream 的“多余处理”

现在又有一个问题出现了，全新的输入输出流对象 SaslInputStream 和 SaslOutputStream，与正常情况的输入输出流对象有什么不同呢？

官方源码对此的解释如下：

Read 读方法从底层输入数据流读入的数据将会被 SaslServer 或 SaslClient 对象进行额

外地处理。

也就是说，所有的数据读写操作需要被 SaslClient 或者 SaslServer 进行额外地处理。从源码中，我们也可以找到这里所指的“额外处理”部分的代码。

首先是 SalsOutputStream 写数据时的处理：

```
public void write(byte[] inBuf, int off, int len) throws IOException {
    if (!useWrap) {
        outputStream.write(inBuf, off, len);
        return;
    }
    try {
        // 此处将会进行额外的包装处理
        if (saslServer != null) {
            saslToken = saslServer.wrap(inBuf, off, len);
        } else {
            saslToken = saslClient.wrap(inBuf, off, len);
        }
    } catch (SaslException se) {
        ...
    }
}
```

然后是 SalsInputStream 读数据时的处理：

```
private int readMoreData() throws IOException {
    ...
    try {
        // 此处进行解包装处理
        if (saslServer != null) {
            obuffer = saslServer.unwrap(saslToken, 0, saslToken.length);
        } else {
            obuffer = saslClient.unwrap(saslToken, 0, saslToken.length);
        }
    } catch (SaslException se) {
        ...
    }
}
```

从上述方法的执行过程来看，这恰好是一次包装、解包装的过程，并不是加、解密的过程。在 jdk 的 SaslServer 类中对 wrap 操作方法的声明解释为：“此方法的结果会组成 Sasl 缓冲区的内容（在 RFC 2222 中定义），并且不包含表示长度的 4 个八位组的前景字段。”这里可以理解为 wrap 方法将会对输入内容进行重新组织保存。

当然如果我们什么安全配置都没开启的话，这些过程都不会发生，还是会按照原来普通的输入、输出流的方式进行数据读写的处理。

2.5.3 BlockToken 认证与 HDFS 的 Sasl 认证对比

以上内容对 HDFS 的 Sasl 只是讲了个大概，内部的诸多细节还是很多的，有兴趣的同学可以自行研究。下面对本节所述的两个认证体系做一下对比。

共同点：

- 没有空间局部的限制，都是数据全局的认证。
- 都会对数据读写效率造成一定程度的影响。

不同点：

- 认证维度不同。BlockToken 认证的粒度较细，是针对块级别的认证，会对每次的块操作做认证。Sasl 则是针对每次数据传输操作做认证。
- 复杂性不同。BlockToken 的认证过程相对简单、清晰。而 Sasl 认证体系则复杂一些，会经过握手阶段，而且中间还可以配置相关的认证防护级别（Qop）的参数。论完整度而言，Sasl 比 BlockToken 更加完整化、体系化一些。

2.6 HDFS 内部的磁盘目录服务

在 HDFS 中，集群的数据分散地存储在各个节点的磁盘上。成千上万个文件、成千上万个磁盘，在某些时刻会很容易发生数据的损坏或节点磁盘的损坏。随着集群规模的扩大，这种事件发生的概率将会越来越高。对此，HDFS 在 DataNode 所在的节点中启动了多种磁盘目录的检测服务，来保证数据的完整性与一致性。这些服务程序并不都是周期性的服务，而是根据其使用的场景做了具体的设计。本节将主要介绍目前现有的三大磁盘检测服务：DiskChecker、DirectoryScanner 和 VolumeScanner。对于每种磁盘检测服务，我们将从使用场景以及作用原理两方面进行详细讲述。

2.6.1 HDFS 的三大磁盘目录检测扫描服务

如前面所提到的，HDFS 为了保证 DataNode 上数据的完整性与一致性，在 DataNode 上启动了三大磁盘目录扫描服务：DiskChecker、DirectoryScanner 和 VolumeScanner。以下是三大服务的简单介绍：

- DiskChecker：坏盘检测服务。检测的级别是每个磁盘，检测的对象是 FsVolume，FsVolume 对应一个存储数据的磁盘。通过检测文件目录的访问权限以及目录是否可创建来判断目录所属磁盘的好坏，如果是坏盘，则此块盘将会被移除，上面的所有块都将被重新复制。
- DirectoryScanner：目录扫描服务，对每块盘上的目录做扫描，使之与内存中维护的块信息同步。比如存储在磁盘上的块已经没有了，则内存中的块信息也应该被移除。
- VolumeScanner：磁盘目录扫描服务。从名称上来看，VolumeScanner 与 DirectoryScanner 比较类似，但是 VolumeScanner 才是真正意义上的块检查服务。它会对已发现的“可疑块”做检查，判断此块是否为损坏块，如果是，则会将其汇报给 NameNode。

以上三大磁盘目录服务对于 `DataNode` 来说,起到了保驾护航的作用,下面在原理和细节上对以上三种服务进行进一步分析。

2.6.2 DiskChecker : 坏盘检测服务

这里的坏盘指的是坏的磁盘。为什么要对坏盘做监控呢?一般用户使用 HDFS 的时候,会将每个 `DataNode` 数据目录所在的本地目录挂载到某块独立的盘上,以此来完全利用节点的存储空间。所以如果某块盘突然发生了硬件故障导致写文件失败,这块盘将会被 HDFS 检测出来,并加入到坏盘列表,其上的数据也将被完全拷贝一份。也就是说, `DataNode` 从此刻开始就完全不会用这块盘了。由此可见, HDFS 对于坏盘的检测还是非常看重的,毕竟谁也不想把数据放在坏了的磁盘上吧。

`DiskChecker` 服务并不是一个周期性的定时任务,它只会在可能有坏盘出现的场景中被启动,然后执行。在这点上,如果你没有仔细研究过它的原理,可能会很容易被它的名称所误解。

1. DiskChecker 何时被调用

首先,要找到 `DiskChecker` 在哪里被真正执行,答案在 `DataNode` 类中。

```
// 启动磁盘错误检查线程方法
private void startCheckDiskErrorThread() {
    checkDiskErrorThread = new Thread(new Runnable() {
        @Override
        public void run() {
            ...
        }
    });
}
```

从上面的代码可以看出, `DiskChecker` 操作是被包装在一个线程当中。然后我们只要继续寻找到 `startCheckDiskErrorThread` 的调用方,就可以知道哪里会调用到磁盘检测服务了。在这里可以找到下面的方法:

```
public void checkDiskErrorAsync() {
    synchronized(checkDiskErrorMutex) {
        checkDiskErrorFlag = true;
        // 如果磁盘检测服务为空
        if(checkDiskErrorThread == null) {
            // 新建磁盘检测服务对象实例
            startCheckDiskErrorThread();
            // 启动此服务,进行磁盘的检测
            checkDiskErrorThread.start();
            LOG.info("Starting CheckDiskError Thread");
        }
    }
}
```

```

    }
}

```

checkDiskErrorAsync 方法就是 DataNode 对外提供的磁盘检测方法。checkDiskErrorAsync 方法有多处调用的场景，这里以其中的一处调用为例：

```

private int receivePacket() throws IOException {
    // 读取下一个数据包
    packetReceiver.receiveNextPacket(in);

    PacketHeader header = packetReceiver.getHeader();
    if (LOG.isDebugEnabled()){
        LOG.debug("Receiving one packet for block " + block +
            ": " + header);
    }

    ...

    final boolean shouldNotWriteChecksum = checksumReceivedLen == 0
        && streams.isTransientStorage();
    try {
        ...
    } catch (IOException iex) {
        // 写数据时发生 IO 异常则启动一次磁盘错误检查
        datanode.checkDiskErrorAsync();
        throw iex;
    }
}

```

在 BlockReceiver 的 receivePacket 方法的 IO 异常处理中，启动了坏盘检测。类似的还有其他调用场景，唯一不同的场景是 FsDatasetImpl 的 validateBlockFile 方法，相关代码如下：

```

File validateBlockFile(String bpid, long blockId) {
    final File f;
    synchronized(this) {
        f = getFile(bpid, blockId, false);
    }

    if(f != null) {
        if(f.exists())
            return f;
        // 如果文件存在，但是实际文件并不存在，则有可能磁盘损坏
        datanode.checkDiskErrorAsync();
    }

    if (LOG.isDebugEnabled()) {
        LOG.debug("blockId=" + blockId + ", f=" + f);
    }
}

```

```

return null;
}

```

所以综合以上情况，DiskChecker 的上游调用流程可以参考图 2-22。

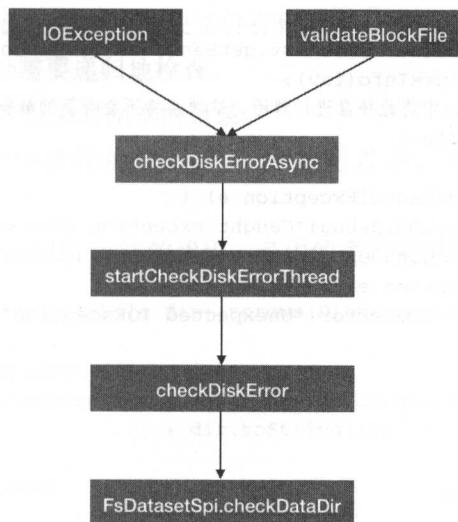


图 2-22 DiskChecker 上游调用

2. DiskChecker 坏盘检测原理

坏盘检测的上游调用已经被我们理清了，但是它内部的执行细节是怎样的呢？在何种情况下，DiskChecker 才会把一块盘视作坏盘呢？

这里进入 checkDataDir 方法：

```

public Set<File> checkDataDir() {
    return volumes.checkDirs();
}

```

进一步查看 checkDirs 方法实现：

```

Set<File> checkDirs() {
    synchronized(checkDirsMutex) {
        Set<File> failedVols = null;
        // 获取当前所有的 volume, volume 对应的是存储此 volume 的盘
        final List<FsVolumeImpl> volumeList = getVolumes();
        // 遍历每个 volume
        for(Iterator<FsVolumeImpl> i = volumeList.iterator(); i.hasNext(); ) {
            final FsVolumeImpl fsv = i.next();
            try (FsVolumeReference ref = fsv.obtainReference()) {
                // 对此 volume 进行检测
                fsv.checkDirs();
            }
        }
    }
}

```

```

    } catch (DiskErrorException e) {
        // 如果在此期间发生 DiskError 异常, 则此块将被移入到坏盘中
        FsDatasetImpl.LOG.warn("Removing failed volume " + fsv + ": ", e);
        if (failedVols == null) {
            failedVols = new HashSet<>(1);
        }
        failedVols.add(new File(fsv.getBasePath()).getAbsolutePath());
        addVolumeFailureInfo(fsv);
        // 在 DataNode 中将此坏盘进行移除, 这块盘将不会有新的数据写入
        removeVolume(fsv);
        ...
    } catch (ClosedChannelException e) {
        FsDatasetImpl.LOG.debug("Caught exception when obtaining " +
            "reference count on closed volume", e);
    } catch (IOException e) {
        FsDatasetImpl.LOG.error("Unexpected IOException", e);
    }
}

...

return failedVols;
}
}

```

继续进入 fsv.checkDirs 的调用:

```

void checkDirs() throws DiskErrorException {
    // TODO:FEDERATION valid synchronization
    // 这里对存在于每个 volume 上的不同的 BlockPool 存储目录做检查
    for(BlockPoolSlice s : bpSlices.values()) {
        s.checkDirs();
    }
}

```

之前的章节中已经提到过, HDFS 是可以拥有多个 BlockPool 的, 这些不同的 BlockPool 在每个盘上的存储是以 BP 打头的目录做区分的, 类似格式如下, 其中 xx.xx.xx.xx 代表的是当时做格式化操作的 NameNode 的 ip 地址:

```
BP-805037254-xx.xx.xx.xx-1460537955319
```

而其对应的相关类是 BlockPoolSlice, 意为每个 BlockPool 在每块目录盘上的分片。重新回到刚才的检查方法 s.checkDirs, 其执行逻辑如下:

```

void checkDirs() throws DiskErrorException {
    // 对 finalized 目录做递归检查
    DiskChecker.checkDirs(finalizedDir);
    // 对 tmp 目录做检查
    DiskChecker.checkDir(tmpDir);
}

```

```
// 对 rbw 目录做检查
DiskChecker.checkDir(rbwDir);
}
```

为什么检查以上 3 大类的目录呢？那就要明白以上 3 类目录的用途关系了：

- finalizedDir 目录，已经完成后的块文件存储目录，层级不止一层，子目录下还存在子目录，所以在此处需要递归地检查。
- tmpDir 临时目录，存储临时副本的目录。
- rbwDir 目录，正在写操作的文件会存放于此目录，写完成之后，会被移入到 finalizedDir 目录中。

下面是 DiskChecker.checkDir 方法的代码，其中的逻辑就是坏盘检测的核心逻辑了：

```
public static void checkDir(File dir) throws DiskErrorException {
    // 尝试创建目录的检查
    if (!mkdirsWithExistsCheck(dir)) {
        throw new DiskErrorException("Cannot create directory: "
            + dir.toString());
    }
    // 文件目录的访问权限的检查
    checkDirAccess(dir);
}
```

在上面的检查逻辑中，包括两大部分的检测。

第一步，创建目录的检测。在这里会通过执行 mkdir 的方法来判断是否能够创建出目录。

```
public static boolean mkdirsWithExistsCheck(File dir) {
    // 尝试创建目录，并检测目录是否存在
    if (dir.mkdir() || dir.exists()) {
        return true;
    }
    // 如果检测失败，则往上一层检测，如果上一层的父目录已不存在，则直接返回 false
    File canonDir = null;
    try {
        canonDir = dir.getCanonicalFile();
    } catch (IOException e) {
        return false;
    }
    String parent = canonDir.getParent();
    return (parent != null) &&
        (mkdirsWithExistsCheck(new File(parent)) &&
            (canonDir.mkdir() || canonDir.exists()));
}
```

第二步，访问权限的检测。检测的逻辑是判断目录的是否能够进行读、写和执行。

```
private static void checkAccessByFileMethods(File dir)
    throws DiskErrorException {
```

```

// 对目录是否可读做检查
if (!FileUtil.canRead(dir)) {
    throw new DiskErrorException("Directory is not readable: "
        + dir.toString());
}
// 对目录是否可写做检查
if (!FileUtil.canWrite(dir)) {
    throw new DiskErrorException("Directory is not writable: "
        + dir.toString());
}
// 对目录是否可执行做检查
if (!FileUtil.canExecute(dir)) {
    throw new DiskErrorException("Directory is not executable: "
        + dir.toString());
}
}

```

坏盘被 DiskChecker 检测出来之后，会在 NameNode 的 50070 端口页面中显示出来，集群管理人员看到了可以做后续的处理工作。

3. DiskChecker 注意点

笔者在工作中发现 DiskChecker 在使用上存在几点需要注意的地方，稍不注意，这些点会给你埋下不少坑。



注意

- 坏盘检测的误判。磁盘损坏毕竟是一个硬件问题，而 DiskChecker 是在软件层面做的检查。一旦我们有不符合 DiskChecker 检测逻辑的行为就会导致坏盘出现。也就是说，我们有时候会人工地导致坏盘的出现，比如在之前的 finalizedDir 目录中，我们删除了大部分的目录，那么过了不久之后，这块盘也会被检测为坏盘，但是实质上这块盘并没有故障。
- 大量的坏盘导致 DataNode 启动的失败。DataNode 对坏盘有一定的容忍数量，如果在 DataNode 启动的时候发现坏盘的数量已超过可容忍数量的时候，会导致启动的失败。可容忍坏盘数量取决于配置项 dfs.datanode.failed.volumes.tolerated。

2.6.3 DirectoryScanner：目录扫描服务

粗看 DirectoryScanner 这个名称，你可能看不出这个服务真正的用途，从源码的注释中我们可以获取此服务用途的详细解释：

```
/**
```

```
 * 阶段性扫描数据存储在目录上的块文件以及块元数据信息文件，
```

* 使其数据与 DataNode 内存中所维护的数据趋向一致。

*/

@InterfaceAudience.Private

public class DirectoryScanner implements Runnable {

大意为阶段性扫描块以及块的元数据文件，使之与 DataNode 内存中维护的数据趋向一致。

DirectoryScanner 内部定义了两类线程池以及扫描间隔时间：

```
public class DirectoryScanner implements Runnable {
    private static final Log LOG = LogFactory.getLog(DirectoryScanner.class);

    private final FsDatasetSpi<?> dataset;
    // 报告产生线程池
    private final ExecutorService reportCompileThreadPool;
    // 主线程池
    private final ScheduledExecutorService masterThread;
    // 扫描间隔时间
    private final long scanPeriodMsecs;
    ...
}
```

DirectoryScanner 会被提交到 masterThread 主线程池上，然后定期执行，从而做到周期性执行：

```
void start() {
    shouldRun = true;
    long offset = DFSUtil.getRandom().nextInt((int) (scanPeriodMsecs/1000L)) *
        1000L; //msec
    long firstScanTime = Time.now() + offset;
    LOG.info("Periodic Directory Tree Verification scan starting at "
        + firstScanTime + " with interval " + scanPeriodMsecs);
    // 将 DirectoryScanner 加入主线程池做定期地执行
    masterThread.scheduleAtFixedRate(this, offset, scanPeriodMsecs,
        TimeUnit.MILLISECONDS);
}
```

接着进入 run 方法：

```
@Override
public void run() {
    try {
        if (!shouldRun) {
            // 停止当前服务，如果此线程已经被外界标记为停止状态
            LOG.warn("this cycle terminating immediately because 'shouldRun' has been
                deactivated");
            return;
        }
        // 执行数据同步操作
        reconcile();
    }
}
```

```

    } catch (Exception e) {
        ...
    }
}

```

继续进入 reconcile 方法：

```

// 同步磁盘和内存中的块数据信息
void reconcile() throws IOException {
    // 扫描内存中的块信息与本地块信息，获取扫描信息的 diff 差异列表
    scan();
    // 遍历 diff 信息列表
    for (Entry<String, LinkedList<ScanInfo>> entry : diffs.entrySet()) {
        String bpid = entry.getKey();
        LinkedList<ScanInfo> diff = entry.getValue();

        for (ScanInfo info : diff) {
            // 进行相应的数据更新操作
            dataset.checkAndUpdate(bpid, info.getBlockId(), info.getBlockFile(),
                                   info.getMetaFile(), info.getVolume());
        }
    }
    if (!retainDiffs) clear();
}

```

reconcile 数据的同步化操作会进行两大操作：scan 和 checkAndUpdate 操作。scan 的目的在于获取 diff 差异列表，checkAndUpdate 才是使数据一致化的操作。

1. scan 生成 diff 差异报告

diff 差异报告的生成需要同时获取磁盘上的块信息报告和内存中的块信息报告，然后做具体维度的比较，比较逻辑代码如下：

```

void scan() {
    clear();
    // 首先获取磁盘上的块报告列表
    Map<String, ScanInfo[]> diskReport = getDiskReport();

    // 此处获取 FSDataset 锁来保证能够同步操作块的映射图对象
    synchronized(dataset) {
        for (Entry<String, ScanInfo[]> entry : diskReport.entrySet()) {
            String bpid = entry.getKey();
            ScanInfo[] blockpoolReport = entry.getValue();

            Stats statsRecord = new Stats(bpid);
            stats.put(bpid, statsRecord);
            LinkedList<ScanInfo> diffRecord = new LinkedList<ScanInfo>();
            // 加入 BlockPool 对应的初始 diff 记录
            diffs.put(bpid, diffRecord);
        }
    }
}

```



```

statsRecord.totalBlocks = blockpoolReport.length;
// 获取内存中的块报告信息
List<FinalizedReplica> bl = dataset.getFinalizedBlocks(bpid);
FinalizedReplica[] memReport = bl.toArray(new FinalizedReplica[bl.size()]);
// 对内存报告进行排序
Arrays.sort(memReport);
// blockpoolReport 对象当前下标
int d = 0;
// memReport 下标
int m = 0;
while (m < memReport.length && d < blockpoolReport.length) {
    FinalizedReplica memBlock = memReport[m];
    ScanInfo info = blockpoolReport[d];
    // 第一种情况, 内存中的块丢失, 而磁盘中的块还在
    if (info.getBlockId() < memBlock.getBlockId()) {
        if (!dataset.isDeletingBlock(bpid, info.getBlockId())) {
            // 块信息在内存中丢失, 进行计数的累加
            statsRecord.missingMemoryBlocks++;
            addDifference(diffRecord, statsRecord, info);
        }
        d++;
        continue;
    }
    // 第二种情况, 磁盘中的块丢失, 而内存中的块还在
    if (info.getBlockId() > memBlock.getBlockId()) {
        addDifference(diffRecord, statsRecord,
            memBlock.getBlockId(), info.getVolume());
        m++;
        continue;
    }
    if (info.getBlockFile() == null) {
        // 第三种情况, 块元数据文件存在, 而块文件不存在
        addDifference(diffRecord, statsRecord, info);
    } else if (info.getGenStamp() != memBlock.getGenerationStamp()
        || info.getBlockFileLength() != memBlock.getNumBytes()) {
        // 第四种情况, 块元数据文件中的版本值或文件长度不一致
        statsRecord.mismatchBlocks++;
        addDifference(diffRecord, statsRecord, info);
    } else if (info.getBlockFile().compareTo(memBlock.getBlockFile()) != 0) {
        // 第五种情况, 块文件对象不一致
        statsRecord.duplicateBlocks++;
        addDifference(diffRecord, statsRecord, info);
    }
    d++;

    if (d < blockpoolReport.length) {
        // 对于同一块, 在多个磁盘上可能会存在记录信息, 此处无须增加内存报告的下标
        ScanInfo nextInfo = blockpoolReport[Math.min(d, blockpoolReport.length - 1)];
        if (nextInfo.getBlockId() != info.getBlockId()) {
            ++m;

```

```

    }
    } else {
        ++m;
    }
}
...
} //end for
} //end synchronized
}

```

主要针对以上 5 种情况的差异做记录,然后将差异记录存入 diffRecord 当中。磁盘上的块相关文件指的是存放在每个数据存储目录下最后一层目录的文件。在 subdir 目录下,存的是块文件以及对应的元数据文件。下面是笔者在测试集群中的一个存储目录:

```

$ ls
blk_1191182458 blk_1191182458_117448171.meta blk_1207959779
blk_1207959779_134228809.meta

```

2. DataNode 的更新操作

以上操作比较完毕之后,就要进行相应数据的更新趋同操作了,也就是下面的 checkAndUpdate 方法:

```

for (ScanInfo info : diff) {
    dataset.checkAndUpdate(bpid, info.getBlockId(), info.getBlockFile(),
        info.getMetaFile(), info.getVolume());
}

```

checkAndUpdate 中的逻辑检查比较复杂,笔者对其进行了部分省略:

```

public void checkAndUpdate(String bpid, long blockId, File diskFile,
    File diskMetaFile, FsVolumeSpi vol) throws IOException {
    Block corruptBlock = null;
    ReplicaInfo memBlockInfo;
    synchronized (this) {
        memBlockInfo = volumeMap.get(bpid, blockId);
        if (memBlockInfo != null && memBlockInfo.getState() != ReplicaState.FINALIZED) {
            // 当前块处于未最终确认状态,此处进行返回操作
            return;
        }

        final long diskGS = diskMetaFile != null && diskMetaFile.exists() ?
            Block.getGenerationStamp(diskMetaFile.getName()) :
            GenerationStamp.GRANDFATHER_GENERATION_STAMP;

        if (diskFile == null || !diskFile.exists()) {
            if (memBlockInfo == null) {
                // 块文件不存在,同时块信息在内存中也不存在,如果此时块元数据文件存在,则进行移除
                if (diskMetaFile != null && diskMetaFile.exists())

```

```

        && diskMetaFile.delete()) {
            LOG.warn("Deleted a metadata file without a block "
                + diskMetaFile.getAbsolutePath());
        }
        return;
    }
    if (!memBlockInfo.getBlockFile().exists()) {
        // 如果磁盘中的块已经不存在了, 但是内存中的块还在的话, 从内存中移除
        volumeMap.remove(bpid, blockId);
        ...
    }
    return;
}

if (memBlockInfo == null) {
    // 磁盘中的块存在, 而内存中的块不存在, 则加入块到内存中的 volumeMap 对象中
    ReplicaInfo diskBlockInfo = new FinalizedReplica(blockId,
        diskFile.length(), diskGS, vol, diskFile.getParentFile());
    volumeMap.add(bpid, diskBlockInfo);
    if (vol.isTransientStorage()) {
        ramDiskReplicaTracker.addReplica(bpid, blockId, (FsVolumeImpl) vol);
    }
    LOG.warn("Added missing block to memory " + diskBlockInfo);
    return;
}

// 文件块的比较
File memFile = memBlockInfo.getBlockFile();
if (memFile.exists()) {
    if (memFile.compareTo(diskFile) != 0) {
        ...
    }
} else {
    ...
}

// 比较版本号
if (memBlockInfo.getGenerationStamp() != diskGS) {
    ...
}

// 如果内存中的块文件信息与块中的文件信息不一致, 则以块文件中的为准
if (memBlockInfo.getNumBytes() != memFile.length()) {
    // 更新文件大小信息
    corruptBlock = new Block(memBlockInfo);
    LOG.warn("Updating size of block " + blockId + " from "
        + memBlockInfo.getNumBytes() + " to " + memFile.length());
    memBlockInfo.setNumBytes(memFile.length());
}
}

// 其他情况将被视为坏块, 并发送给 NameNode

```

```

if (corruptBlock != null) {
    LOG.warn("Reporting the block " + corruptBlock
        + " as corrupt due to length mismatch");
    try {
        datanode.reportBadBlocks(new ExtendedBlock(bpid, corruptBlock));
    } catch (IOException e) {
        LOG.warn("Failed to report bad block " + corruptBlock, e);
    }
}
}
}

```

DirectoryScanner 是一项周期性的服务，默认间隔执行时间 6 小时。DirectoryScanner 像是一个“园丁”的角色，将这段时间内出现的一些异常的数据处理掉，维护内存中的数据与磁盘上块数据的完整性与一致性。此过程流程图如图 2-23 所示。

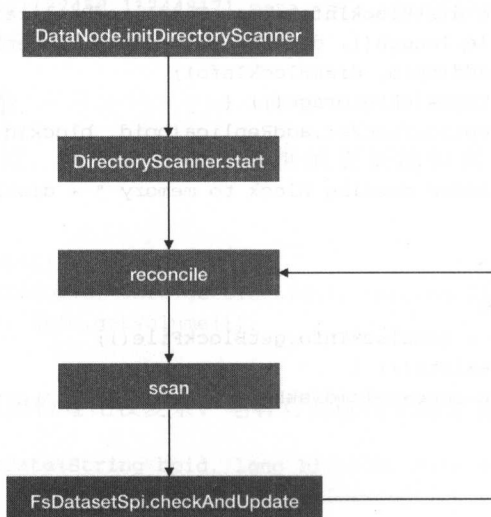


图 2-23 DirectoryScanner 运行流程

2.6.4 VolumeScanner：磁盘目录扫描服务

VolumeScanner 是专门针对每块磁盘做块扫描的服务。块扫描类似于一次健康状况的检查。每个 VolumeScanner 有属于它自己的独立线程，代码如下：

```

/**
 * 每个 VolumeScanner 扫描一块盘，并且每个 VolumeScanner 有其独自の线程，
 * 它们统一被 DataNode 的 BlockScanner 对象所管理。
 */
public class VolumeScanner extends Thread

```

如上面注释中所说明的，VolumeScanner 是被 BlockScanner 所管理的。同时 VolumeScanner

的初始化也是在 BlockScanner 中进行的。在 BlockScanner 的 addVolumeScanner 中会进行新的 VolumeScanner 创建与启动:

```
public synchronized void addVolumeScanner(FsVolumeReference ref) {
    boolean success = false;
    try {
        ...
        LOG.debug("Adding scanner for volume {} (StorageID {})",
            volume.getBasePath(), volume.getStorageID());
        // 为此 volume 所代表的磁盘新建 VolumeScanner 对象
        scanner = new VolumeScanner(conf, datanode, ref);
        // 启动此线程
        scanner.start();
        // 加入到 VolumScanner 映射图中
        scanners.put(volume.getStorageID(), scanner);
        success = true;
    } finally {
        if (!success) {
            // 如果之前创建 VolumeScanner 对象没有成功, 则不需要此引用对象
            // 将其清除即可
            IOUtils.cleanup(null, ref);
        }
    }
}
```

下面直接进入 VolumeScanner 的 run 方法:

```
public void run() {
    // 记录扫描开始的时间
    this.startMinute =
        TimeUnit.MINUTES.convert(Time.monotonicNow(), TimeUnit.MILLISECONDS);
    this.curMinute = startMinute;
    try {
        LOG.trace("{}: thread starting.", this);
        // 初始化处理类
        resultHandler.setup(this);
        try {
            long timeout = 0;
            while (true) {
                ExtendedBlock suspectBlock = null;
                // 获取当前对象的对象锁, 以此保证操作可疑块列表的同步性
                synchronized (this) {
                    ...
                    // 获取下一个可疑块
                    suspectBlock = popNextSuspectBlock();
                }
                // 进行可疑块的扫描
                timeout = runLoop(suspectBlock);
            }
        } catch (InterruptedException e) {
            ...
        }
    }
}
```

在 run 方法中执行的逻辑如下：

- 1) 初始化块处理类。
- 2) 循环获取下一个可疑检查块。
- 3) 检测扫描选出的可疑块。

上面代码中的 resultHandler 处理类会在扫描块操作的时候被调用。这里的 while 循环是块扫描的主要操作，在这个循环中，会进行可疑块的筛选和扫描操作。下面先来看 popNextSuspectBlock 的处理逻辑：

```
private synchronized ExtendedBlock popNextSuspectBlock() {
    Iterator<ExtendedBlock> iter = suspectBlocks.iterator();
    if (!iter.hasNext()) {
        return null;
    }
    // 从可疑块列表中选出块
    ExtendedBlock block = iter.next();
    iter.remove();
    return block;
}
```

这里的可疑块是从 suspectBlocks 块列表中移出的，而 suspectBlocks 是 VolumeScanner 所维护的可疑块列表：

```
// 可疑块列表，scanner 线程将会优先扫描这些块
private final LinkedHashSet<ExtendedBlock> suspectBlocks =
    new LinkedHashSet<ExtendedBlock>();
```

经过调用发现，将块标记为“可疑块”从而将其加入到可疑块列表中：

```
public synchronized void markSuspectBlock(ExtendedBlock block) {
    if (stopping) {
        LOG.info("{}: Not scheduling suspect block {} for " +
            "rescanning, because this volume scanner is stopping.", this, block);
        return;
    }
    ...
    if (suspectBlocks.contains(block)) {
        LOG.info("{}: suspect block {} is already queued for " +
            "rescanning.", this, block);
        return;
    }
    // 如果可疑块列表中不存在此块，则加入列表
    suspectBlocks.add(block);
    recentSuspectBlocks.put(block, true);
    LOG.info("{}: Scheduling suspect block {} for rescanning.", this, block);
    notify(); // 唤醒 scanner 线程
}
```

同样我们需要找到 `markSuspectBlock` 的调用场景，如下：

```
private int sendPacket(ByteBuffer pkt, int maxChunks, OutputStream out,
    boolean transferTo, DataTransferThrottler throttler) throws IOException {
    ...

    int dataOff = checksumOff + checksumDataLen;
    if (!transferTo) {
        ...
    }

    try {
        if (transferTo) {
            ...
        } catch (IOException e) {
            if (e instanceof SocketTimeoutException) {
                ...
                datanode.metrics.incrSendDataPacketTimeout();
            } else {
                ...
                // 发生其他的 IOException 的时候，标记此块为可疑块
                datanode.getBlockScanner().markSuspectBlock(
                    volumeRef.getVolume().getStorageID(),
                    block);
                datanode.metrics.incrSendDataPacketExceptionNum();
            }
            throw ioeToSocketException(e);
        }
    }

    ...
    return dataLen;
}
```

在 `BlockSender` 的 `sendPacket` 读数据方法中，如果发生了 IO 异常，则会进行可疑块的标记。这个与 `DiskChecker` 类似，都是在异常的场景中进行触发处理。可疑块被筛选出来之后，会经过 `runLoop` 方法的处理：

```
private long runLoop(ExtendedBlock suspectBlock) {
    long bytesScanned = -1;
    boolean scanError = false;
    ExtendedBlock block = null;
    try {
        long monotonicMs = Time.monotonicNow();
        expireOldScannedBytesRecords(monotonicMs);

        if (!calculateShouldScan(volume.getStorageID(), conf.targetBytesPerSec,
            scannedBytesSum, startMinute, curMinute)) {
            // 如果扫描块的带宽速率设置得大小导致不允许进行扫描操作，则返回需要等待的时间
            return 30000L;
        }
    }
```

```

// 寻找一个可用的 BlockPool 准备扫描
if (suspectBlock != null) {
    // 如果当前可疑块不为空, 则设置为当前扫描的块
    block = suspectBlock;
} else {
    // 否则从 BlockPool 中选出一个待扫描的块
    if ((curBlockIter == null) || curBlockIter.atEnd()) {
        // 如果当前 BlockPool 已经扫描到头了, 则选取下一个 BlockPool
        long timeout = findNextUsableBlockIter();
        ...
    }
    try {
        // 获取当前 BlockPool 中的下一个待扫描块
        block = curBlockIter.nextBlock();
    } catch (IOException e) {
        // 选取块时出现异常, 打出异常信息警告
        LOG.warn("{}: nextBlock error on {}", this, curBlockIter);
        ...
        return 0L;
    }
    ...
    // 对选定好的块, 进行扫描, 同时传入带宽速率进行限流
    bytesScanned = scanBlock(block, conf.targetBytesPerSec);
    ...
}
}

```

以上的处理逻辑主要表达了两个意思:

- 1) 从 suspectBlock (可疑块列表) 中选出一个待扫描块进行扫描。
- 2) 如果 suspectBlock 没有可选的块了, 则从 BlockPool 中进行选取, 如果当前的 BlockPool 中的块已经遍历完毕, 则选取下一个 BlockPool, 继续遍历其中的块。

选定块后, 接着会进行 scanBlock 方法:

```

private long scanBlock(ExtendedBlock cblock, long bytesPerSec) {
    ExtendedBlock block = null;
    ...
    if (block == null) {
        // 返回 -1 表示块没有找到
        return -1;
    }
    BlockSender blockSender = null;
    try {
        blockSender = new BlockSender(block, 0, -1,
            false, true, true, datanode, null,
            CachingStrategy.newDropBehind());
        // 用 BlockSender 对象读块的时候, 对其做了限流操作
        throttler.setBandwidth(bytesPerSec);
        long bytesRead = blockSender.sendBlock(nullStream, null, throttler);
        // 进行结果的处理
    }
}

```



```

        resultHandler.handle(block, null);
        return bytesRead;
    } catch (IOException e) {
        // 异常时同样进行结果的处理
        resultHandler.handle(block, e);
    } finally {
        IOUtils.cleanup(null, blockSender);
    }
    return -1;
}

```

BlockSender 在扫描块的时候，特意对其进行了限流，防止其对 DataNode 正常读写的影响。在 resultHandler 的 handle 处理方法中，包含了对扫描块的最终处理：

```

public void handle(ExtendedBlock block, IOException e) {
    FsVolumeSpi volume = scanner.volume;
    // 如果没有发生 IOException 异常，则意味着块是正常块
    if (e == null) {
        LOG.trace("Successfully scanned {} on {}", block, volume.getBasePath());
        return;
    }
    // 如果块不存在于 DataNode 上，同样返回
    if (!volume.getDataset().contains(block)) {
        LOG.debug("Volume {}: block {} is no longer in the dataset.",
            volume.getBasePath(), block);
        return;
    }
    ...
    if (e instanceof FileNotFoundException) {
        LOG.info("Volume {}: verification failed for {} because of " +
            "FileNotFoundException. This may be due to a race with write.",
            volume.getBasePath(), block);
        return;
    }
    LOG.warn("Reporting bad {} on {}", block, volume.getBasePath());
    // 其他异常情况，则视此块为坏块，并且上报坏块
    try {
        scanner.datanode.reportBadBlocks(block);
    } catch (IOException ie) {
        // 汇报坏块失败的时候，同样打出异常信息
        LOG.warn("Cannot report bad " + block.getBlockId(), e);
    }
}

```

resultHandler 的核心逻辑是根据 BlockSender 读块时是否抛出 IO 异常来作为块好坏的评判标准。以上处理逻辑流程见图 2-24。

以上就是 HDFS 在 DataNode 中所启动的三大服务，每个服务都有特定的作用。熟悉了这些服务，将会使我们对 DataNode 的运行以及故障问题排查有很大的帮助。

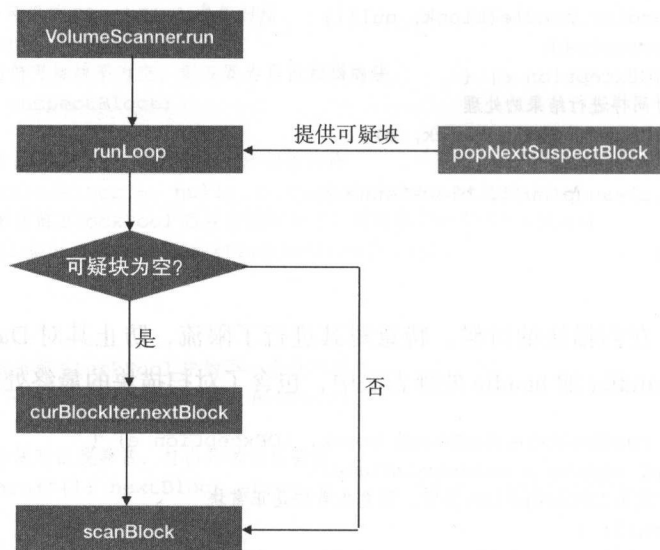


图 2-24 VolumeScanner 运行过程

2.7 小结

本章内容覆盖的范围比较广，而且部分章节的内容可能会比较难懂，比如 HDFS 的快照管理，HDFS 如何进行缓存块的管理等等。当然还有一些是偏重理解性的内容，阅读完本章节我们要大致理解 HDFS 内部的 Sasl 认证过程以及其与 BlockToken 验证的差别所在，还有 DataNode 内部的数据保护服务等等。经典的三副本策略机制也是我们需要了解的内容，副本放置位置的选择是其中比较重要的一点。

HDFS 的新颖功能特性

本章介绍 HDFS 中的一些“非主流”的功能特性，这些功能并不是不好，而是说它很少被人用到，容易被大家忽视。本章首先介绍 HDFS 的视图文件系统 ViewFileSystem。谁说 HDFS 上的路径是一成不变的？ViewFs 可以让你随意变更存储的路径名称。其次介绍 WebHDFS，WebHDFS 的出现可以使得用户使用 HDFS 的成本变得更为简单。第三介绍 HDFS 数据加密空间 Encryption zone，在加密空间下，你看到的将只会是一堆加密的数据流。第四介绍 HDFS EC 纠删码技术，这是用户非常期待的一个特性，它能够解决当下由于三副本备份策略导致的存储空间浪费的问题。第五介绍 HDFS 对象存储技术 Ozone，它使得用户将数据存入到 HDFS 中变得更容易。

3.1 HDFS 视图文件系统：ViewFileSystem

HDFS 视图文件系统并不是一个全新类型的文件系统，正如它的名称所描述的，它只是一个“视图”。“视图”的意思代表它只是在表面上做了改变的文件系统，而真实指向的文件系统其实并没有发生变化。进一步来说，HDFS 视图文件系统可以跨越多个集群，保持文件系统在逻辑上的一致性。视图文件系统在 HDFS 中的名称叫作 ViewFileSystem，简称为 ViewFs。在本节中，我们将主要介绍视图文件系统的使用场景、实现原理和配置的使用三方面内容。

为了突出视图文件系统的作用，这里先来讨论一下传统数据合并的方案。在数据合并中，常用的做法是搬迁数据。例如在 HDFS 中，我们会想到用 DistCp 工具进行远程拷贝。

虽然 DistCp 本身就是用来干这种事情的,但是随着数据量规模的升级,会有以下问题的出现:

- 拷贝周期太长,如果数据量非常大,在机房总带宽有限的情况下,拷贝的时间将会非常长。
- 数据在拷贝的过程中,一定会有原始数据的变更与改动,如何同步这些数据也是需要考虑的方面。

以上两点,是笔者想到的比较突出的两个问题。而本文所要讲述的 ViewFileSystem 则能很巧妙地解决此问题。

3.1.1 ViewFileSystem: 视图文件系统

关于 ViewFileSystem 的概念,首先要明白一个核心原则:ViewFileSystem 不是一个新的文件系统,只是逻辑上的一个视图文件系统,在逻辑上是唯一的。

这句话怎么理解呢,ViewFileSystem 就是帮大家做了一件事情:将各个集群的真实文件路径与 ViewFileSystem 内新定义的路径进行关联映射。

上面这句话的意思就好比文件系统中挂载的意思。进一步地说,ViewFileSystem 会在每个客户端中维护一份挂载关系表,就是上面说的集群物理路径->视图文件系统路径这样的指向关系。但是在挂载关系表中,关系当然不止一个,会有很多个。比如下面所示的多对关系:

```
/user      -> hdfs://nn1/containingUserDir/user
/project/foo -> hdfs://nn2/projects/foo
/project/bar -> hdfs://nn3/projects/bar
```

前面是 ViewFileSystem 中的路径,后者才是代表的真正集群路径。所以你可以理解为 ViewFileSystem 真正干的事情是路径的路由解析。图 3-1 是简单的原理图。

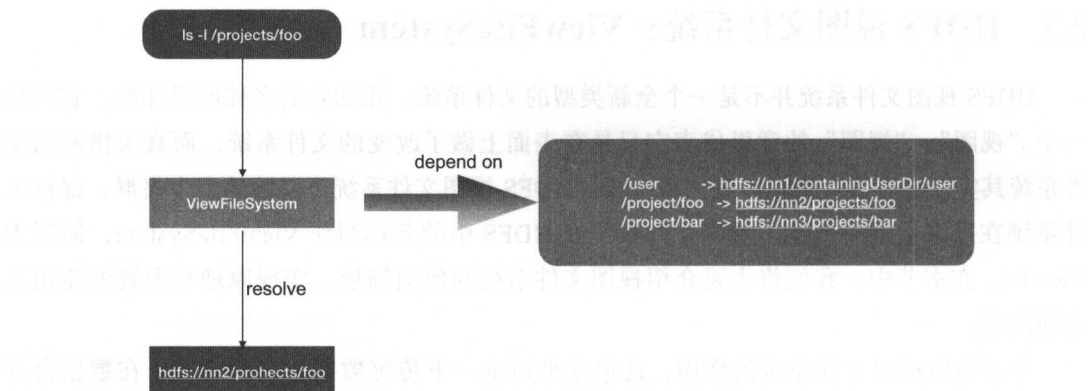


图 3-1 ViewFileSystem 的路由解析

3.1.2 ViewFileSystem 内部实现原理

在上文中我们已经基本了解到 ViewFileSystem 的作用是一个路由解析的角色，真实的请求处理还是在各自真实的集群上。这小节探讨的内容是 ViewFileSystem 内部是如何实现这个“路由解析”的角色。

1. 目录挂载点

因为要做的是路由解析，所以挂载点的设计就显得非常重要了。下面来看 ViewFileSystem 中对挂载点的定义：

```
static public class MountPoint {
    // 源路径
    private Path src;
    // 目录指向路径，也就是真实路径，可以为多个
    private URI[] targets;
    MountPoint(Path srcPath, URI[] targetURIs) {
        src = srcPath;
        targets = targetURIs;
    }
    Path getSrc() {
        return src;
    }
    URI[] getTargets() {
        return targets;
    }
}
```

一般情况下，挂载节点是一对一的。但是如果存在不同集群间有相同名称目录的情况，也是可以进行一对多的，在 Hadoop 中叫做 MergeCount，不过这个功能目前尚未完成，还在开发中，相关 JIRA：HADOOP-8298（ViewFs merge mounts）。

2. 挂载点的解析与存放

在 ViewFileSystem 初始化操作中，挂载点的解析与存放是其中一个关键的过程。其中的过程执行是在下面这个变量中进行的：

```
// 此对象可理解挂载表
InodeTree<FileSystem> fsState;
```

进入 ViewFileSystem 的初始化方法：

```
public void initialize(final URI theUri, final Configuration conf)
    throws IOException {
    super.initialize(theUri, conf);
    setConf(conf);
    config = conf;
    // 根据配置信息，初始化挂载表实例
```

```

final String authority = theUri.getAuthority();
try {
    myUri = new URI(FsConstants.VIEWFS_SCHEME, authority, "/", null, null);
    // 传入 conf 配置信息进行 fsState 初始化
    fsState = new InodeTree<FileSystem>(conf, authority) {
        ...
    };
}

```

然后进入 InodeTree 的构造方法中：

```

protected InodeTree(final Configuration config, final String viewName)
    throws UnsupportedOperationException, URISyntaxException,
    ...

final String mtPrefix = Constants.CONFIG_VIEWFS_PREFIX + "." +
    vName + ".";
final String linkPrefix = Constants.CONFIG_VIEWFS_LINK + ".";
final String linkMergePrefix = Constants.CONFIG_VIEWFS_LINK_MERGE + ".";
boolean gotMountTableEntry = false;
final UserGroupInformation ugi = UserGroupInformation.getCurrentUser();
for (Entry<String, String> si : config) {
    final String key = si.getKey();
    // 判断源 key 名是否以前缀 fs.viewfs.mounttable 开头
    if (key.startsWith(mtPrefix)) {
        ...
        // 获取目标映射的真实路径，可能为多个，以“,”隔开
        final String target = si.getValue();
        createLink(src, target, isMergeLink, ugi);
    }
}
...

```

真正实现挂载点关系的存储是在 createLink 方法中，代码如下：

```

private void createLink(final String src, final String target,
    final boolean isLinkMerge, final UserGroupInformation aUgi)
    throws URISyntaxException, IOException,
    FileAlreadyExistsException, UnsupportedOperationException {
    // 验证源路径是否为有效的路径
    final Path srcPath = new Path(src);
    if (!srcPath.isAbsoluteAndSchemeAuthorityNull()) {
        throw new IOException("ViewFs:Non absolute mount name in config:" + src);
    }

    // 将待添加的路径按照“/”分隔符进行拆分
    final String[] srcPaths = breakIntoPathComponents(src);
    // 设置当前节点为根节点
    INodeDir<T> curInode = root;
    ...
}

```

注意上面执行的最后一行代码，出现了 INodeDir 类，而且设置了当前 curInode 为根节点。这实际上是非常有用意的。INodeDir 类的定义为：

```
// 此类代表挂载表中的一个挂载目录
static class INodeDir<T> extends INode<T> {
    // 孩子节点
    final Map<String, INode<T>> children = new HashMap<String, INode<T>>();
    // 与此挂载目录相关的文件系统
    T InodeDirFs = null;
    boolean isRoot = false;
    ...
}
```

从上面我们可以看出这里是一个父亲 - 孩子的关系。每个目录会有对应自身的目标文件系统，而且孩子可能还是 `INodeDir` 或是 `INode` 的子类。因为路径按照符号“/”进行划分，我们大致可以推测出 `ViewFileSystem` 是按照树型结构的存放方式进行挂载点的存储的。

下面的代码基本上验证了上面的猜想，下面是目录树的查找过程：

```
...
int i;
// 忽略第一个空字符串，遍历其后的每一个子段
for (i = 1; i < srcPaths.length-1; i++) {
    // 获取当前的子段字符串
    final String iPath = srcPaths[i];
    // 从当前的目录 INode 中进行查找
    INode<T> nextInode = curInode.resolveInternal(iPath);
    // 如果没有查找到，意味着当前的节点中没有此路径下对应的信息
    if (nextInode == null) {
        // 新增此路径对应的目标文件系统的信息
        INodeDir<T> newDir = curInode.addDir(iPath, aUgi);
        newDir.InodeDirFs = getTargetFileSystem(newDir);
        // 并以此作为下个节点，即为查找到的目标节点
        nextInode = newDir;
    }
    // 如果此节点已经是 INodeLink 信息，则抛异常
    if (nextInode instanceof INodeLink) {
        throw new FileAlreadyExistsException("Path " + nextInode.fullPath +
            " already exists as link");
    } else {
        // 如果还是 INode 目录，则将子目录作为当前目录，往下寻找
        assert(nextInode instanceof INodeDir);
        curInode = (INodeDir<T>) nextInode;
    }
}
...
}
```

找到最近一层的目录树后，在此加入新 URI 的 Link 关联信息

```
...
// 到此基本找到了最底层的目录，然后在此目录下添加 INodeLink 链接
final INodeLink<T> newLink;
final String fullPath = curInode.fullPath + (curInode == root ? "" : "/" )
    + iPath;
```

```

// 将目录 URL 链接等信息全部传入 INodeLink 中
if (isLinkMerge) {
    String[] targetsList = StringUtils.getStrings(target);
    URI[] targetsListURI = new URI[targetsList.length];
    int k = 0;
    for (String itarget : targetsList) {
        targetsListURI[k++] = new URI(itarget);
    }
    newLink = new INodeLink<T>(fullPath, aUgi,
        getTargetFileSystem(targetsListURI), targetsListURI);
} else {
    newLink = new INodeLink<T>(fullPath, aUgi,
        getTargetFileSystem(new URI(target)), new URI(target));
}
// 将构造完毕的 INodeLink 作为子节点加入到当前节点中
curInode.addLink(iPath, newLink);
// 同时加入到挂载节点列表中
mountPoints.add(new MountPoint<T>(src, newLink));
...

```

可以看到，最终指向的文件系统和具体信息都在 INodeLink 中。所有的挂载目录点的位置都以字符串的形式被树形地拆开存放。换句话说，在 ViewFileSystem 中输入一个 ViewFileSystem 中配置的查询路径，会被逐层解析到对应的 INodeDir。图 3-2 为 INodeLink 存储结构图。

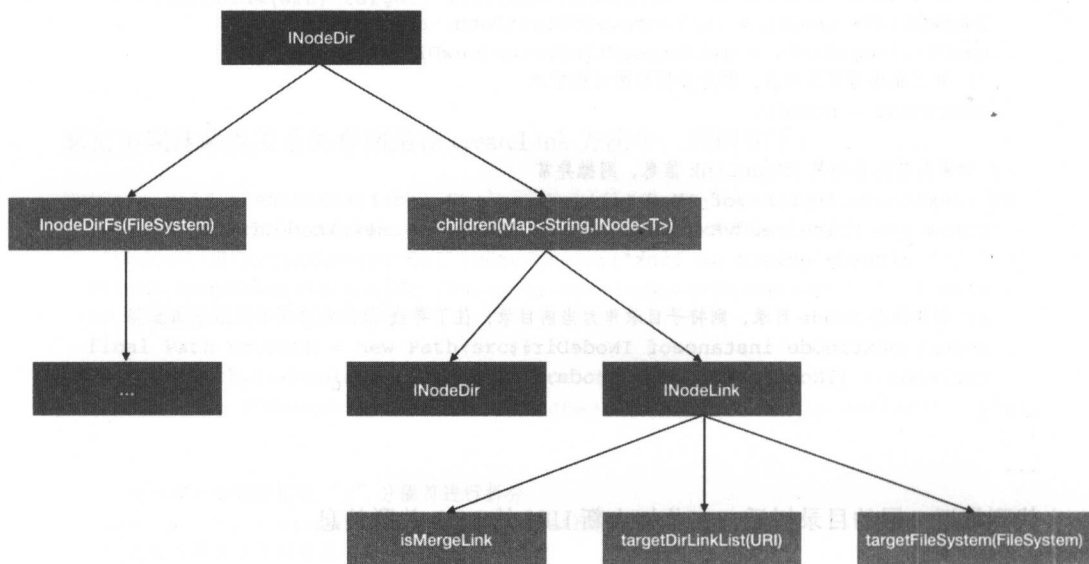


图 3-2 INodeDir 存储结构

解析的逻辑与此完全类似，也是通过 INodeDir 的存储关系一步一步往里找，这里就不展开介绍了。

3. ViewFileSystem 的请求处理

现在又有一个问题出现了，ViewFileSystem 是如何处理客户端发来的各种文件操作请求的呢？以 mkdir 为例：

```
@Override
public boolean mkdirs(final Path dir, final FsPermission permission)
    throws IOException {
    // 通过 fsState 对象进行解析
    InodeTree.ResolveResult<FileSystem> res =
        fsState.resolve(getUriPath(dir), false);
    // 获取目标真实文件系统进行对应的请求处理
    return res.targetFileSystem.mkdirs(res.remainingPath, permission);
}
```

这里的 fsState.resolve 就会到之前提到的挂载点树中进行逐层寻找。找到对应的文件系统后，就会把后面最终起作用的路径作为参数传入真实的文件系统中。

此过程的调用流程见图 3-3。

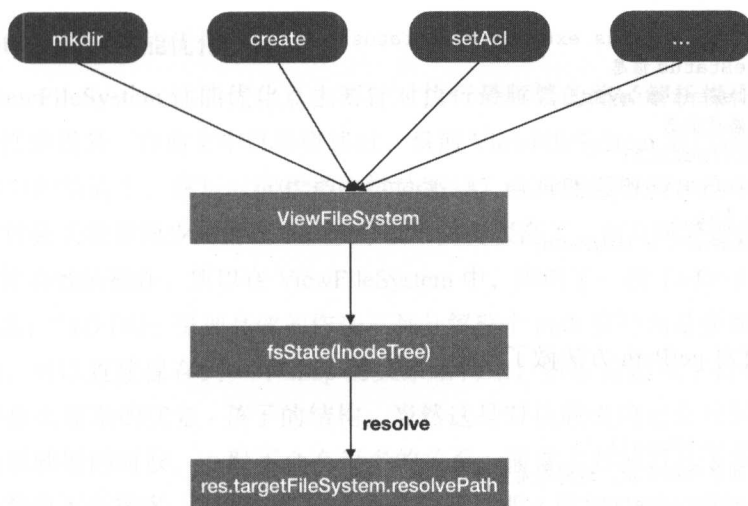


图 3-3 ViewFs 的路径解析

4. ViewFileSystem 的路径包装

ViewFileSystem 作为一个视图文件系统，要保持在逻辑上的完全一致，需要对文件返回的属性信息做一层包装和适配。

例如，笔者事先设定了挂载信息：

```
/project/viewFsTmp -> hdfs://nn1/projects/Tmp
```

前者是笔者设定的 ViewFileSystem 路径，后者是真实文件系统存放路径。在真实文件

系统中假设存在 3 个子文件：

```
/projects/Tmp/child1
/projects/Tmp/child2
/projects/Tmp/child3
```

在 ViewFileSystem 的情况下，我们用 `hadoop fs -ls /project/viewFsTmp` 的命令去看，出现的信息应该是这样的：

```
/projects/viewFsTmp/child1
/projects/viewFsTmp/child2
/projects/viewFsTmp/child3
```

因为挂载点信息文件路径已经被变更了，一切都得按照 ViewFileSystem 中所配置的路径来，所以这需要对真实返回的 FileStatus 文件信息对象做一层包装。但对于其他一些如大小、修改时间等基本属性信息，直接返回原来的信息就行。而对于一些 Path 的返回，就要做一层修改了。

于是就衍生出了 ViewFsFileStatus 这个类：

```
class ViewFsFileStatus extends FileStatus {
    // 原 FileStatus 信息
    final FileStatus myFs;
    // 修改的路径信息
    Path modifiedPath;
    ViewFsFileStatus(FileStatus fs, Path newPath) {
        myFs = fs;
        modifiedPath = newPath;
    }
    ...
}
```

在此类中就对 getPath 方法做了改动：

```
@Override
public Path getPath() {
    // 重载返回路径的方法，返回的是修改过的路径信息
    return modifiedPath;
}
```

对于其他基本属性方法，直接调用原来的方法即可。

```
...
// 获取副本数量信息
@Override
public short getReplication() {
    // 调用实际指向的文件系统对象获取副本信息，后续方法同理
    return myFs.getReplication();
}
```

```
@Override
public long getModificationTime() {
    return myFs.getModificationTime();
}
```

```
@Override
public long getAccessTime() {
    return myFs.getAccessTime();
}
```

```
...
```

这就实现了我们前面所描述的例子。这一点，大家可以在自己的环境中进行测试。

图 3-4 是 ViewFsFileStatus 继承关系图。

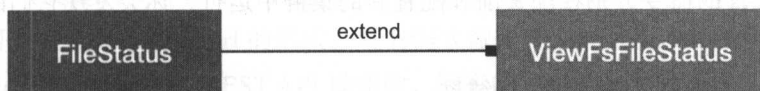


图 3-4 ViewFsFileStatus 的继承关系

5. ViewFileSystem 性能优化

这里的 ViewFileSystem 性能优化点主要针对执行最频繁的路径解析操作，这里其实可以有比较大的性能提升。在前文中已经讲述过，目前 ViewFileSystem 通过将路径进行拆分，保存到一个树型的结构中，解析时同样需要拆分路径，进行逐层解析。这个方法本身没有问题，但是在挂载表数量很少的情况下，效率就不见得很高了。而且频繁地拆分、合并字符串本身不是非常高效的操作。所以在 ViewFileSystem 中，声明了一段 To-Do 的声明，就是将来可以做的改进：“TO DO：更加高效的作法不是分解整个 path 路径而是做简单的比较。”

也就是说，可以直接保存到一个 map 的关系结构中，然后直接做字符串的简单比较，也不需要保存什么复杂的父亲 - 孩子的结构。当然这是对挂载表内记录比较少少的情况。而我们选择目录映射的时候，一般不会有更多的关系，通常主要映射几个顶级目录就可以了，所以记录数也不会很多。当然如果将来改进的话，那么添加 Link 和解析路径的逻辑都要变。

3.1.3 ViewFileSystem 的使用

最后简单介绍 ViewFileSystem 的配置使用，主要分为两个步骤。

第一步，创建 Viewfs 名称。

在 core-site.xml 中配置 fs.defaultFS 属性，如下所示：

```
<property>
  <name>fs.defaultFS</name>
```

```
<value>viewfs://MultipleCluster</value>
</property>
```

第二步，添加挂载关系：

```
<property>
<name>fs.viewfs.mounttable.MultipleCluster.link./viewfstmp</name>
<value>hdfs://nn1/tmp</value>
</property>
```

这里的 nn1 就是真实的集群路径。注意这里的 fs.viewfs.mounttable.MultipleCluster.link 中 link 前面的名称必须与之前 Viewfs 中定义的名称一致。

经过这 2 个步骤，就基本完成了 ViewFileSystem 的配置，其实非常简单。

然后用 fs -ls 的命令分别在配置前和配置后的集群中运行，你会发现它们的子目录文件信息完全一致。

配置前在 nn1 所在集群内的执行结果：

```
$ hadoop fs -ls /tmp
Found 2 items

-rw-r--r--  2 data      supergroup  193488274  2016-04-13  14:21  /tmp/share.tar.gz
drwxr-xr-x  - data      supergroup           0  2016-03-15  15:39  /tmp/sparkjars
```

配置后集群中的执行结果：

```
$ hadoop fs -ls /viewfstmp
Found 2 items

-rw-r--r--  2 data      supergroup  193488274  2016-04-13  14:21  /viewfstmp/share.tar.gz
drwxr-xr-x  - data      supergroup           0  2016-03-15  15:39  /viewfstmp/sparkjars
```

倘若你在配置后的集群中还是用 hadoop fs -ls /tmp 去查找，它会报找不到文件的提示。

```
$ hadoop fs -ls /tmp
ls: `/tmp': No such file or directory
```

因为在逻辑上，这个目录已经被挂载到 Viewfs 的 /viewfstmp 路径下了。这些挂载信息会维护在客户端的内存中，不需要重启 NameNode 和 DataNode。

3.2 HDFS 的 Web 文件系统：WebHdfsFileSystem

在 HDFS 中，如果用户想要对 HDFS 中的文件或目录做操作，他需要了解 NameNode 对外开发的方法，然后调用 DFSCClient 对应的方法。对于一个天天使用 HDFS 的人来说，这非常容易。但是对于之前一点都没有接触过 HDFS 的人来说，还是需要一定的学

习成本的。对于这种情况，HDFS 内部提供的 WebHdfsFileSystem 能够很好地解决这个问题，而且它能够帮助用户很容易地将 HDFS 中的文件、目录操作结合到自己的系统中。WebHdfsFileSystem 让 HDFS 以 Web 的形式展现给用户，用户通过调用相应的 REST API 即可完成文件、目录的操作。本节主要的目标是让大家了解以及使用 WebHdfsFileSystem，具体的内容包括 HDFS Web 文件系统 REST API 的设计、流程调用以及使用方法等。本节部分内容来自于 Hadoop 社区官方文档。

3.2.1 WebHdfsFileSystem 的 REST API 操作

WebHdfsFileSystem 是 HDFS 以 Web 的形式呈现出的一种文件系统。既然是 Web 形式，所以它有很关键的一个特性：通过 Web REST API 的形式对文件系统进行操作。也就是说，用户可以通过 http 请求 url 的形式，对 HDFS 做操作，并且不用写任何客户端程序。

在 WebHdfsFileSystem 的 REST API 操作中，所有的方法被分为以下 4 类：

- ❑ GET
- ❑ PUT
- ❑ POST
- ❑ DELETE

对于不同操作性质的方法，需要用相应的 REST API 类型。比如获取状态信息之类的方法，就用 GET 请求的方式。在 WebHdfsFileSystem 中，同样对自身的操作方法进行了类型划分。下面是具体的划分结果：

1) GET 方式。

获取文件目录信息相关：

- ❑ OPEN
- ❑ GET_FILE_STATUS
- ❑ LIST_STATUS
- ❑ GET_CONTENT_SUMMARY
- ❑ GET_FILE_CHECKSUM
- ❑ GET_HOME_DIRECTORY
- ❑ GET_BLOCK_LOCATIONS

获取属性、ACL 访问相关：

- ❑ GET_DELEGATION_TOKEN
- ❑ GET_XATTRS
- ❑ LIST_XATTRS
- ❑ GET_ACL_STATUS

☐ CHECK_ACCESS

2) PUT 方式。

文件目录设置相关:

☐ CREATE

☐ MKDIRS

☐ CREATE_SYMLINK

☐ RENAME

☐ SET_REPLICATION

☐ SET_OWNER

☐ SET_PERMISSION

☐ SET_TIMES

ACL 访问属性相关:

☐ RENEW_DELEGATION_TOKEN

☐ CANCEL_DELEGATION_TOKEN

☐ MODIFY_ACL_ENTRIES

☐ REMOVE_ACL_ENTRIES

☐ REMOVE_DEFAULT_ACL

☐ REMOVE_ACL

☐ SET_ACL

☐ SET_XATTR

☐ REMOVE_XATTR

快照操作相关:

☐ CREATE_SNAPSHOT

☐ RENAME_SNAPSHOT

3) POST 方式:

☐ APPEND: 文件追加操作

☐ CONCAT: 文件拼接操作

☐ TRUNCATE: 文件截断操作

4) DELETE 方式:

☐ DELETE: 文件 / 目录删除操作

☐ DELETE_SNAPSHOT: 快照删除操作

3.2.2 WebHdfsFileSystem 的流程调用

在这里我们以一个简单的请求为例，来模拟 WebHdfsFileSystem 的整体调用过程。比如下面的 rename 操作，调用命令如下：

```
curl -i -X PUT "<HOST>:<PORT>/webhdfs/v1/<PATH>?op=RENAME&destination=<PATH>"
```

这里利用 curl 命令来发送请求。

首先这个请求会调用到 HDFS 客户端的 WebHdfsFileSystem 类中的对应方法：

```
public boolean rename(final Path src, final Path dst) throws IOException {
    statistics.incrementWriteOps(1);
    // 构造 rename 操作类型
    final HttpOpParam.Op op = PutOpParam.Op.RENAME;
    // 调用 FsPathBooleanRunner 执行器进行请求的进一步处理
    return new FsPathBooleanRunner(op, src,
        new DestinationParam(makeQualified(dst).toUri().getPath())
    ).run();
}
```

FsPathBooleanRunner 具体如何执行，后面会具体介绍，这里先跳过。然后 FsPathBooleanRunner 的 run 方法将会调用 HDFS 服务端的 NamenodeWebHdfsMethods 类，这个类包含了对应请求类型的处理方法。

```
private Response put(
    final UserGroupInformation ugi,
    final DelegationParam delegation,
    ...
) throws IOException, URISyntaxException {

    final Configuration conf = (Configuration)context.getAttribute(JspHelper.CURRENT_
        CONF);
    final NameNode namenode = (NameNode)context.getAttribute("name.node");
    final NamenodeProtocols np = getRPCServer(namenode);

    switch(op.getValue()) {
    case CREATE:
        //...
    case MKDIRS:
        ...
    case CREATESYMLINK:
        ...
    // RENAME 操作对应的处理
    case RENAME:
    {
        final EnumSet<Options.Rename> s = renameOptions.getValue();
        // 执行对应的重命名方法
        if (s.isEmpty()) {
            final boolean b = np.rename(fullpath, destination.getValue());
```

```

        final String js = JsonUtil.toJsonString("boolean", b);
        return Response.ok(js).type(MediaType.APPLICATION_JSON).build();
    } else {
        np.rename2(fullpath, destination.getValue(),
            s.toArray(new Options.Rename[s.size()]));
        return Response.ok().type(MediaType.APPLICATION_OCTET_STREAM).build();
    }
}
...

```

最后 `np.rename` 会触发对 `FSNamesystem` 的调用，也就最终完成了整个过程的调用。整个流程调用过程见图 3-5。

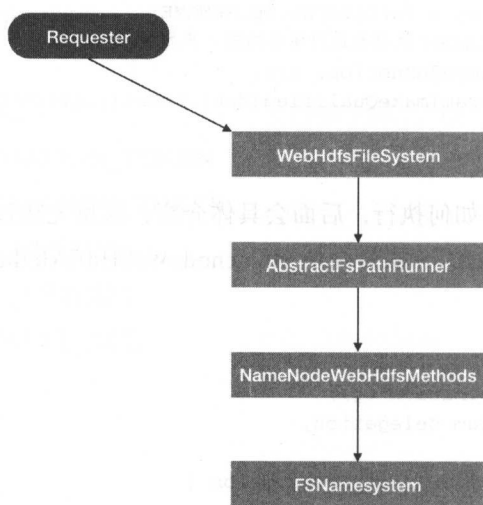


图 3-5 WebHdfsFileSystem 执行过程

3.2.3 WebHdfsFileSystem 执行器调用

前面 `rename` 命令的调用过程会经过 `FsPathBooleanRunner` 执行器的处理。`FsPathBooleanRunner` 本身是 `AbstractRunner` 的子类，而 `AbstractRunner` 在其内部会做许多事情，包括：

- 1) 初始化 http 请求连接。
- 2) 连接 Server 端。
- 3) 获取执行回复内容。
- 4) 执行失败重试操作。

从这里可以看出，`AbstractRunner` 为客户端免去了一些公共的实现细节。下面进入 `run` 方法，查看其内部是如何执行这些操作的。


```

T run() throws IOException {
    // 获取当前执行用户信息
    UserGroupInformation connectUgi = ugi.getRealUser();
    if (connectUgi == null) {
        connectUgi = ugi;
    }
    if (op.getRequireAuth()) {
        connectUgi.checkTGTAndReloginFromKeytab();
    }
    try {
        // 连接的过程需要运行在 doAs 方法内，以此确保中间认证过程的正确执行
        return connectUgi.doAs(
            new PrivilegedExceptionAction<T>() {
                @Override
                public T run() throws IOException {
                    // 以重试策略的方式执行
                    return runWithRetry();
                }
            });
    } catch (InterruptedException e) {
        throw new IOException(e);
    }
}

```

runWithRetry 方法的代码如下：

```

private T runWithRetry() throws IOException {
    ...
    // 重新计数从 0 开始
    for(int retry = 0; ; retry++) {
        checkRetry = !redirected;
        final URL url = getUrl();
        try {
            // 根据 url 建立 http 连接
            final HttpURLConnection conn = connect(url);
            // 输出流在关闭的时候需要进行验证
            if (!op.getDoOutput()) {
                validateResponse(op, conn, false);
            }
            // 获取执行回复结果
            return getResponse(conn);
        } catch (AccessControlException ace) {
            ...
        } catch (IOException ioe) {
            // 发生异常时，判断是否需要重试
            shouldRetry(ioe, retry);
        }
    }
}

```

当连接出现失败、NameNode 进入 Standby 模式或 Server 端执行出错，都会导致 IO 异

常的发生。这里的 `getResponse` 是抽象方法，它会根据具体子类需要的回复格式返回相应格式的内容。在一般的 http 请求回复中，大多有两种类型的回复方式：第一种是带有具体信息内容的回复，一般以 json 格式进行返回；第二种，不带具体信息内容的回复。在 WebHDFS 的操作响应中，总共分出了以下 4 种类型的响应回复：

1) `FsPathResponseRunner`：带 json 回复格式的处理执行器。这类执行器一般用于 Get 类型的操作，比如 `getHdfsFileStatus` 操作。

2) `FsPathRunner`：不带 json 回复格式的处理执行器，它的 `getResponse` 返回内容为空，如下所示：

```
// 基于路径实现，并且无 json 回复的处理类
class FsPathRunner extends AbstractFsPathRunner<Void> {
    ...
    @Override
    Void getResponse(URLConnection conn) throws IOException {
        return null;
    }
}
```

像 `rename`、`setXAttr` 这样的操作作用的就是 `FsPathRunner` 执行器。

3) `FsPathConnectionRunner`：此执行器的回复是 http 的连接对象。

```
class FsPathConnectionRunner extends AbstractFsPathRunner<URLConnection> {
    ...
    @Override
    URLConnection getResponse(final URLConnection conn)
        throws IOException {
        return conn;
    }
}
```

4) `FsPathOutputStreamRunner`：用以处理创建、追加数据流。

`WebHdfsFileSystem` 中的执行器结构关系见图 3-6。

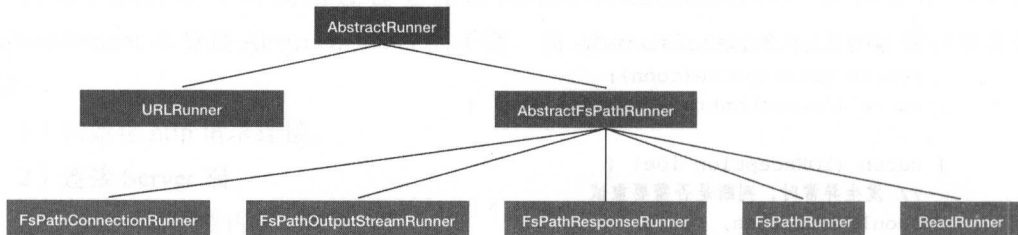


图 3-6 `WebHdfsFileSystem` 的执行器继承关系

3.2.4 WebHDFS 的 OAuth2 认证

WebHDFS REST API 的出现,大大简化了用户使用 HDFS 的成本,用户只需通过构造简单的请求链接就可以达到操作 HDFS 文件目录的效果。但同时另外一个问题也随之而来,那就是安全问题。过于简单的操作方式会导致操作的误用和恶意使用。用户只需要知道目标集群 NameNode 的 Ip 和目标操作文件路径,就可以对其进行直接操作,这至少听上去还是比较恐怖的。因此我们需要在 WebHDFS 内部增加一些认证操作来规避这样的现象。在 WebHDFS 的认证方面,社区已经有了一定的进展,HDFS-8155 (Support OAuth2 in WebHDFS) 已经完成了在 WebHDFS 上的 OAuth2 认证支持。

1. OAuth2 认证机制

OAuth2 是 OAuth 协议的下一版本。与 OAuth1.0 相比,OAuth2 在客户端的使用上,更加简单、易用。

OAuth2 认证将会涉及以下 3 个角色:

- 服务端: 用户使用服务端提供的各个资源。
- 用户: 服务端资源的真实拥有者。
- 客户端: 要访问服务端资源的第三方应用。

下面是具体的认证步骤:

- 1) 用户登录客户端向服务端请求一个临时令牌。
- 2) 服务端通过客户端验证后,返回其临时令牌。
- 3) 客户端获取临时令牌后,引导用户到服务端提供的授权页面。
- 4) 用户输入帐号、密码后,表明用户授权此客户端访问所请求的资源。
- 5) 授权过程完成后,客户端根据临时令牌从服务端获取访问令牌。
- 6) 客户端获取访问令牌后向服务端访问受保护的资源。

比较常见的使用 OAuth2 认证的一个场景是 QQ 的授权认证。

2. WebHdfsFileSystem 的 OAuth2 认证调用

WebHdfsFileSystem 的 OAuth2 认证调用首先是在 initialize 方法中进行的:

```
public synchronized void initialize(Uri uri, Configuration conf)
    throws IOException {
    super.initialize(uri, conf);
    setConf(conf);
    // 设置用户相关配置参数
    UserParam.setUserPattern(conf.get(
        HdfsClientConfigKeys.DFS_WEBHDFS_USER_PATTERN_KEY,
        HdfsClientConfigKeys.DFS_WEBHDFS_USER_PATTERN_DEFAULT));
    // 获取是否启用 WebHdfs 的 OAuth2 认证, 默认不启用
```

```

boolean isOAuth = conf.getBoolean(
    HdfsClientConfigKeys.DFS_WEBHDFS_OAUTH_ENABLED_KEY,
    HdfsClientConfigKeys.DFS_WEBHDFS_OAUTH_ENABLED_DEFAULT);

if(isOAuth) {
    LOG.debug("Enabling OAuth2 in WebHDFS");
    // 如果启用了 OAuth2 的认证, 则构建 OAuth2 连接构造器
    connectionFactory = URLConnectionFactory
        .newOAuth2URLConnectionFactory(conf);
} else {
    // 否则构建默认连接器
    LOG.debug("Not enabling OAuth2 in WebHDFS");
    connectionFactory = URLConnectionFactory
        .newDefaultURLConnectionFactory(conf);
}
...

```

继续进入 newOAuth2URLConnectionFactory 方法:

```

public static URLConnectionFactory newOAuth2URLConnectionFactory(
    Configuration conf) throws IOException {
    ConnectionConfigurator conn;
    try {
        ConnectionConfigurator sslConnConfigurator
            = newSslConnConfigurator(DEFAULT_SOCKET_TIMEOUT, conf);
        // 构造 OAuth2 连接配置器
        conn = new OAuth2ConnectionConfigurator(conf, sslConnConfigurator);
    } catch (Exception e) {
        throw new IOException("Unable to load OAuth2 connection factory.", e);
    }
    return new URLConnectionFactory(conn);
}

```

OAuth2ConnectionConfigurator 在构造函数中会创建 accessTokenProvider 令牌提供实例对象。

```

public OAuth2ConnectionConfigurator(Configuration conf,
    ConnectionConfigurator sslConfigurator) {
    this.sslConfigurator = sslConfigurator;

    notNull(conf, ACCESS_TOKEN_PROVIDER_KEY);

    Class accessTokenProviderClass = conf.getClass(ACCESS_TOKEN_PROVIDER_KEY,
        ConfCredentialBasedAccessTokenProvider.class,
        AccessTokenProvider.class);
    // 在连接配置器中, 构建了 AccessTokenProvider 访问令牌提供实例
    accessTokenProvider = (AccessTokenProvider) ReflectionUtils
        .newInstance(accessTokenProviderClass, conf);
    accessTokenProvider.setConf(conf);
}

```

之后 OAuth2ConnectionConfigurator 连接器会在配置连接的时候使用 OAuth2 的令牌。

```
public HttpURLConnection configure(HttpURLConnection conn)
    throws IOException {
    if(sslConfigurator != null) {
        sslConfigurator.configure(conn);
    }
    // 在每次构建连接时, 会从 AccessTokenProvider 中获取访问令牌
    String accessToken = accessTokenProvider.getAccessToken();
    // 加入头信息中
    conn.setRequestProperty("AUTHORIZATION", HEADER + accessToken);

    return conn;
}
```

AccessTokenProvider 在 WebHDFS 中有不同的实现类, 具体实现类读者可以自行查阅 org.apache.hadoop.hdfs.web.oauth2 包下以 AccessTokenProvider 名称结尾的相关类。

3.2.5 WebHDFS 的使用

WebHDFS 的使用可以通过 curl 命令来操作, 比如下面的 rename 命令:

```
curl -i -X PUT "<HOST>:<PORT>/webhdfs/v1/<PATH>?op=RENAME&destination=<PATH>"
```

-X 后面代表的是请求类型。如果我们在请求中没有额外设置用户, 可能会报出权限拒绝的错误:

```
{ "RemoteException": { "exception": "AccessControlException", "javaClassName": "org.apache.hadoop.security.AccessControlException", "message": "Permission denied: user=dr.who, access=WRITE, inode=\"/tmp-2\":data:supergroup:drwxr-xr-x" } }
```

可以通过添加 user.data=<USER> 的方式来表明当前操作的用户。

```
curl -i -X PUT "<HOST>:<PORT>/webhdfs/v1/<PATH>?user.name=<USER>op=RENAME&destination=<PATH>"
```

如果执行成功, 客户端将会得到成功的回复信息:

```
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{"boolean": true}
```

更多有关 WebHDFS 使用的例子可以阅读 Hadoop 官网的例子。

3.3 HDFS 数据加密空间: Encryption zone

在之前第 2.5 节中我们介绍了 HDFS 内部的两套认证体系,在一定程度上能起到数据保护的作用。在本节中,我们将要介绍另外一套机制:数据加密空间(Encryption zone)。Encryption zone 与之前的 BlockToken 验证相比,它更加注重于空间的特点,因为它只会对指定路径空间下的数据文件,做加、解密操作。本节将主要介绍 Encryption zone 的实现原理以及它的配置使用两方面内容。

3.3.1 Encryption zone 原理介绍

HDFS Encryption zone 加密空间是一种端到端的加密模式。其中的加、解密过程对于客户端来说是完全透明的。数据在客户端读操作的时候被解密,当数据被客户端写的时候进行加密,所以 HDFS 本身并不是一个主要的参与者。形象地说,在 HDFS 中你看到的只是一堆加密的数据流。

了解 HDFS 数据加密空间的原理对于我们使用 Encryption zone 有很大帮助。Encryption zone 是 HDFS 中的一个抽象概念,它表示在此空间下的数据在写的时候会被透明地加密,同时在读的时候,被透明地解密,这是它的核心所在。以下具体到细小的细节:

- 1) 每个 encryption zone 会与每个 encryption zone key 相关联,而这个 key 会在创建 encryption zone 的时候被指定。

- 2) 每个 encryption zone 中的文件会有其唯一的 data encryption key (数据加密 key),简称 DEK。

- 3) DEK 不会被 HDFS 直接处理,HDFS 只处理经过加密的 DEK,叫做 encrypted data encryption key,缩写就是 EDEK。

- 4) 客户端请求 KMS 服务去解密 EDEK,然后利用解密后得到的 DEK 去读、写数据。

在第四步中有一个很重要的过程:在客户端向 KMS 服务请求的时候,会有相关权限的验证,不符合要求的客户端将不会得到解密后的 DEK。而且 KMS 的权限验证是独立于 HDFS 的,是自身的一套权限验证体系。

图 3-7 是 Encryption zone 的原理图。

Key Provider 可以理解为一个 key store 的保存库,其中 KMS 是其中的一个实现。

3.3.2 Encryption zone 源码实现

本小节我们将从源码的层面对上述原理做跟踪分析,从两大方向进行分析:一个是创建文件,并且写文件数据;另一个是读文件数据。

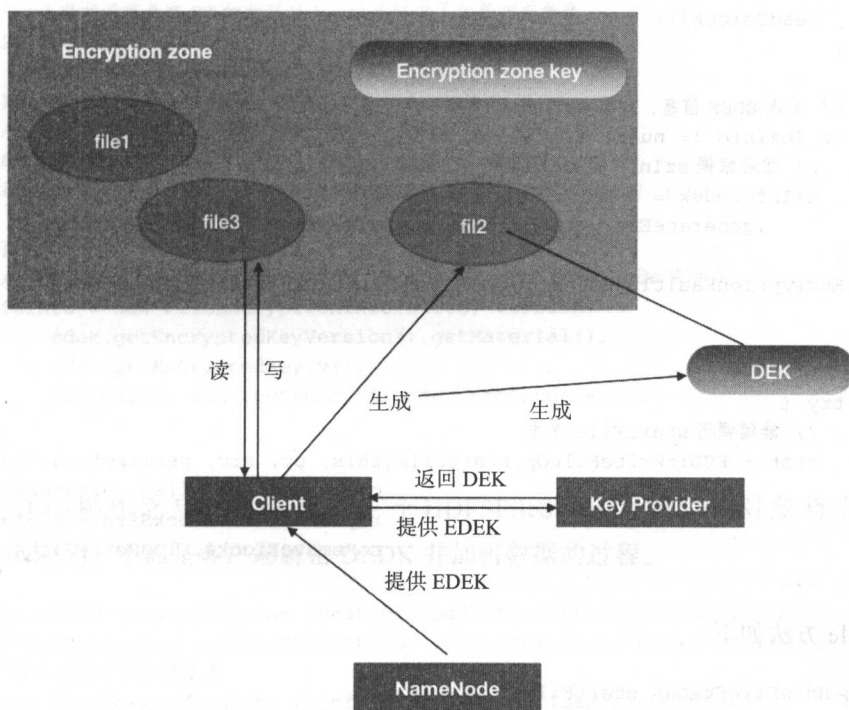


图 3-7 Encryption zone 原理图

1. Encryption zone 下的写文件

首先客户端发起创建文件请求，到了 NameNode 这边，会调用 startFile 方法，用于生成 DEK 和 EDEK：

```

private HdfsFileStatus startFileInt(final String src,
    PermissionStatus permissions, String holder, String clientMachine,
    EnumSet<CreateFlag> flag, boolean createParent, short replication,
    long blockSize, CryptoProtocolVersion[] supportedVersions,
    boolean logRetryCache)
    throws IOException {
    ...
    FSDirWriteFileOp.EncryptionKeyInfo ezInfo = null;
    // 判断 key provider 是否为空
    if (provider != null) {
        readLock();
        try {
            checkOperation(OperationCategory.READ);
            // 不为空，就生成 EncryptionKey info
            ezInfo = FSDirWriteFileOp
                .getEncryptionKeyInfo(this, pc, src, supportedVersions);
        } finally {
    
```

```

        readUnlock();
    }

    // 生成 EDEK 信息, 如果 ezInfo 不为空
    if (ezInfo != null) {
        // 然后根据 ezInfo 的 key 名称生成 EDEK 信息到 ezInfo 中
        ezInfo.edek = FSDirEncryptionZoneOp
            .generateEncryptedDataEncryptionKey(dir, ezInfo.ezKeyName);
    }
    EncryptionFaultInjector.getInstance().startFileAfterGenerateKey();
}

...
try {
    // 继续调用 startFile 方法
    stat = FSDirWriteFileOp.startFile(this, pc, src, permissions, holder,
                                        clientMachine, flag, createParent,
                                        replication, blockSize, ezInfo,
                                        toRemoveBlocks, logRetryCache);
    ...
}

```

startFile 方法如下:

```

static HdfsFileStatus startFile(
    FSNamesystem fsn, FSPermissionChecker pc, String src,
    PermissionStatus permissions, String holder, String clientMachine,
    EnumSet<CreateFlag> flag, boolean createParent,
    short replication, long blockSize,
    EncryptionKeyInfo ezInfo, INode.BlocksMapUpdateInfo toRemoveBlocks,
    boolean logRetryEntry)
    throws IOException {
    assert fsn.hasWriteLock();
    ...

    CipherSuite suite = null;
    CryptoProtocolVersion version = null;
    KeyProviderCryptoExtension.EncryptedKeyVersion edek = null;

    // 取出 ezInfo 中的关键信息
    if (ezInfo != null) {
        edek = ezInfo.edek;
        suite = ezInfo.suite;
        version = ezInfo.protocolVersion;
    }

    ...

    FileEncryptionInfo feInfo = null;

    final EncryptionZone zone = FSDirEncryptionZoneOp.getEZForPath(fsd, iip);
    if (zone != null) {

```



```

// 此路径目前是在 EZ 加密空间下, 但是缺少了加密相关参数
if (suite == null || edek == null) {
    throw new RetryStartFileException();
}
// 如果前面条件都满足了, 则进行 EDEK 的匹配
final String ezKeyName = zone.getKeyName();
if (!ezKeyName.equals(edek.getEncryptionKeyName())) {
    throw new RetryStartFileException();
}
// 传入到 FileEncryptionInfo 中, feInfo 将会被设置到 INode 文件中
feInfo = new FileEncryptionInfo(suite, version,
    edek.getEncryptedKeyVersion().getMaterial(),
    edek.getEncryptedKeyIv(),
    ezKeyName, edek.getEncryptionKeyVersionName());
}
...

```

完成这些操作之后, 将会返回一个 HDFSFileStatus 对象, 此对象将会被 DFSOutputStream 利用。下面是客户端解密 DEDK 并加密数据的过程。

```

public HdfsDataOutputStream createWrappedOutputStream(DFSOutputStream dfsos,
    FileSystem.Statistics statistics, long startPos) throws IOException {
    // 取出文件中的加密信息
    final FileEncryptionInfo feInfo = dfsos.getFileEncryptionInfo();
    if (feInfo != null) {
        // 文件是被加密的, 需要包装数据流为加密流
        getCryptoProtocolVersion(feInfo);
        final CryptoCodec codec = getCryptoCodec(conf, feInfo);
        // 解密 feInfo 中的 EDEK 的信息, 其中会向 KerProvider 进行请求
        KeyVersion decrypted = decryptEncryptedDataEncryptionKey(feInfo);
        // 然后将解密后的信息作为参数, 构造出加密输出流
        final CryptoOutputStream cryptoOut =
            new CryptoOutputStream(dfsos, codec,
                decrypted.getMaterial(), feInfo.getIV(), startPos);
        return new HdfsDataOutputStream(cryptoOut, statistics, startPos);
    } else {
        // 没有 FileEncryptionInfo 信息代表无须进行加密
        return new HdfsDataOutputStream(dfsos, statistics, startPos);
    }
}

```

继续查看 decryptEncryptedDataEncryptionKey 方法, 验证其中是否有向 KeyProvider 请求服务的操作。

```

private KeyVersion decryptEncryptedDataEncryptionKey(FileEncryptionInfo
    feInfo) throws IOException {
    try (TraceScope ignored = tracer.newScope("decryptEDEK")) {
        // 获取 keyProvider 服务实例
        KeyProvider provider = getKeyProvider();
        if (provider == null) {

```

```

        throw new IOException("No KeyProvider is configured, cannot access" +
            " an encrypted file");
    }
    // 获取加密的 key version
    EncryptedKeyVersion ekv = EncryptedKeyVersion.createForDecryption(
        feInfo.getKeyName(), feInfo.getEzKeyVersionName(), feInfo.getIV(),
        feInfo.getEncryptedDataEncryptionKey());
    try {
        KeyProviderCryptoExtension cryptoProvider = KeyProviderCryptoExtension
            .createKeyProviderCryptoExtension(provider);
        // 进行解密操作
        return cryptoProvider.decryptEncryptedKey(ekv);
    } catch (GeneralSecurityException e) {
        throw new IOException(e);
    }
}
}
}

```

构造完加密输出流对象 `CryptoOutputStream` 之后, 在随后的写操作中, 数据都会额外经过一步加密算法的操作。此部分的过程调用见图 3-8。

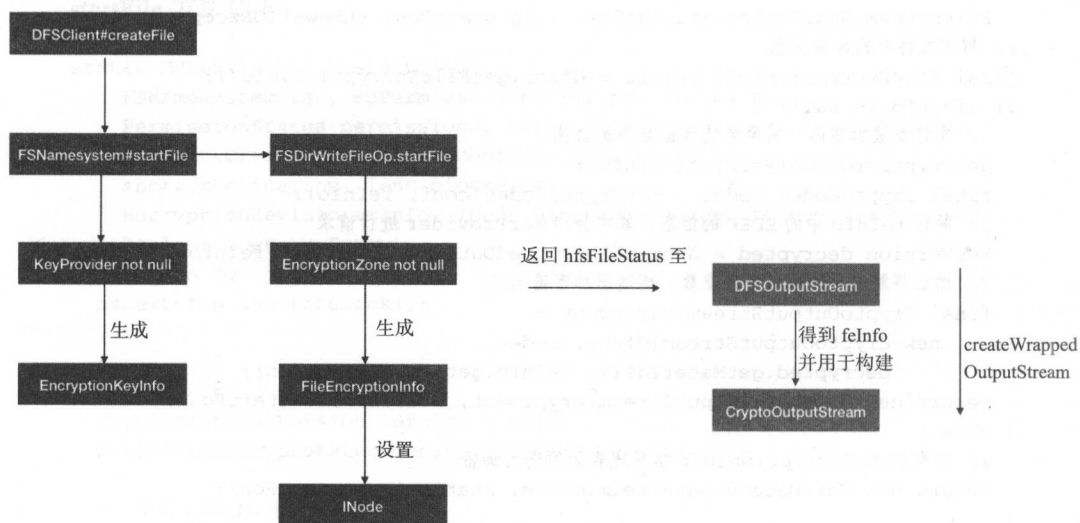


图 3-8 Encryption zone 下的写文件过程

2. Encryption zone 下的读文件

读文件部分的操作与写文件非常类似。首先构造出目标文件的 `HDFSFileStatus` 对象, 然后取出其中的 `FileEncryptionInfo`。在此过程中 `FileEncryptionInfo` 会被设置到 `LocatedBlocks` 中。

```

private static HdfsLocatedFileStatus createLocatedFileStatus(

```

```

FSDirectory fsd, byte[] path, INodeAttributes nodeAttrs,
byte storagePolicy, int snapshot,
boolean isRawPath, INodesInPath iip) throws IOException {

```

```

...
// 然后设置到 LocatedBlocks 中
loc = fsd.getBlockManager().createLocatedBlocks(
    fileNode.getBlocks(snapshot), fileSize, isUc, 0L, size, false,
    inSnapshot, feInfo, ecPolicy);
...

```

之后这些块信息会以参数的形式传入到 `DFSInputStream`，并在方法 `fetchLocatedBlocksAndGetLastBlockLength` 中被设置到相应变量中去。

```

private long fetchLocatedBlocksAndGetLastBlockLength(boolean refresh)
    throws IOException {
    LocatedBlocks newInfo = locatedBlocks;
    ...
    // 将 locatedBlocks 中的 EncryptionInfo 信息设置到变量中
    fileEncryptionInfo = locatedBlocks.getFileEncryptionInfo();

    return lastBlockBeingWrittenLength;
}

```

此信息同样会被取出用到加密输入流中。

```

public HdfsDataInputStream createWrappedInputStream(DFSInputStream dfsis)
    throws IOException {
    // 获取文件加密信息
    final FileEncryptionInfo feInfo = dfsis.getFileEncryptionInfo();
    if (feInfo != null) {
        // 文件是被加密的，包装数据流为加密数据流
        getCryptoProtocolVersion(feInfo);
        final CryptoCodec codec = getCryptoCodec(conf, feInfo);
        // 解密 DEDK
        final KeyVersion decrypted = decryptEncryptedDataEncryptionKey(feInfo);
        // 构造加密输入流
        final CryptoInputStream cryptoIn =
            new CryptoInputStream(dfsis, codec, decrypted.getMaterial(),
                feInfo.getIV());
        return new HdfsDataInputStream(cryptoIn);
    } else {
        // 没有 FileEncryptionInfo 信息，无须进行加密操作
        return new HdfsDataInputStream(dfsis);
    }
}

```

与之前的过程非常类似，在加密输入流中，对读取的数据进行解密，使得用户能看到正常的数据。图 3-9 为此过程图。

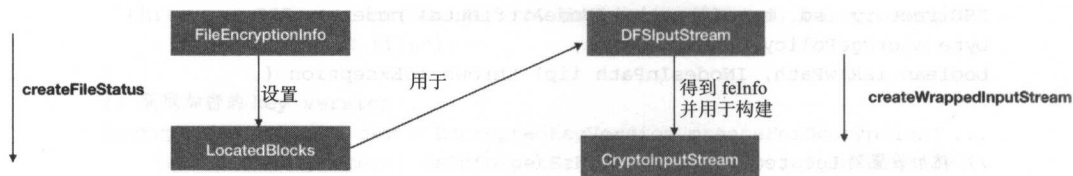


图 3-9 Encryption zone 下的读文件过程

3. Encryption zone 的管理

在源码分析的最后部分，再简单聊聊 Encryption zone 的管理，对应的管理类叫做 EncryptionZoneManager。但是有一点需要注意，它内部维护的对象不是 EncryptionZone，而是 EncryptionZoneInt。

```

public class EncryptionZoneManager {

    public static Logger LOG = LoggerFactory.getLogger(EncryptionZoneManager
        .class);

    ...
    // 用 TreeMap 保存的 Encryption zone 列表
    private final TreeMap<Long, EncryptionZoneInt> encryptionZones;
    private final FSDirectory dir;
    private final int maxListEncryptionZonesResponses;
    ...
  
```

这里 TreeMap 的 key 位置保存的是 Encryption zone 对应目录的 INodeId。EncryptionZoneInt 与 EncryptionZone 有什么微妙的关系呢？在具体使用的时候，EncryptionZoneInt 会被用来构造 EncryptionZone，代码如下：

```

EncryptionZone getEZINodeForPath(INodesInPath iip) {
    final EncryptionZoneInt ezi = getEncryptionZoneForPath(iip);
    if (ezi == null) {
        return null;
    } else {
        // 此处利用 EncryptionZoneInt 对象信息构造 EncryptionZone 对象
        return new EncryptionZone(ezi.getInodeId(), getFullPathName(ezi),
            ezi.getSuite(), ezi.getVersion(), ezi.getKeyName());
    }
}
  
```

通过判断目标路径是否存在于 Encryption zone 列表中，来判断此文件是否为加密文件，然后以 INodeId 作为 key 从 map 中取出。

Encryption zone 的结构管理见图 3-10。

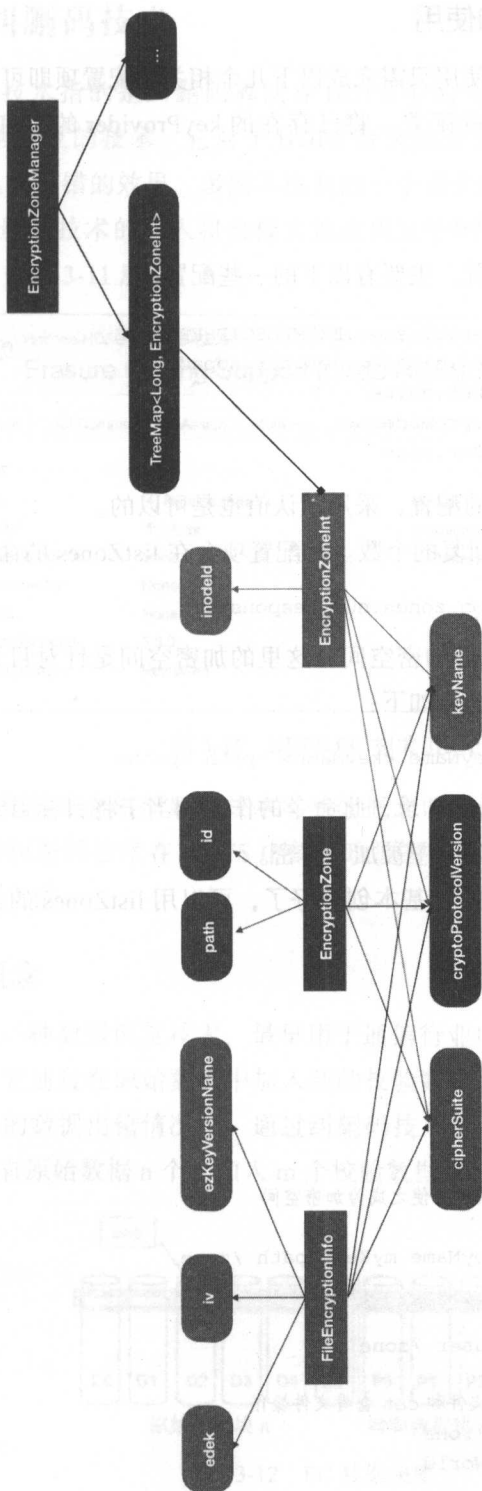


图 3-10 Encryption zone 管理类结构图

3.3.3 Encryption zone 的使用

Encryption zone 功能的配置使用只需完成以下几个相关的配置项即可。

第一步：完成 keyProvider 的配置。将已存在的 keyProvider 的 URL 地址配置到下面的配置项中：

```
dfs.encryption.key.provider.uri
```

第二步：加密算法相关的配置。主要有以下的一些配置项：

```
hadoop.security.crypto.codec.classes.EXAMPLECIPHERSUITE
hadoop.security.crypto.codec.classes.aes.ctr.nopadding
hadoop.security.crypto.cipher.suite
hadoop.security.crypto.jce.provider
hadoop.security.crypto.buffer.size
```

这些配置项并不需要强制性的配置，采用默认值也是可以的。

第三步：配置 listZone 响应回复的个数。此配置项会在 listZones 的命令中起到作用。

```
dfs.namenode.list.encryption.zones.num.responses
```

第四步：创建 Encryption zone 加密空间。这里的加密空间是针对目录级别的，同时还需要设置一个 key 名称，使用的命令如下：

```
hdfs crypto -createZone -keyName <keyName> -path <path>
```

这里的 path 必须是已经建好的目录，此命令的作用相当于将目标目录作为一个加密空间，在此目录下的文件在读写的过程中被加、解密。

以上操作完成之后，加密空间就基本创建好了，可以用 listZones 的命令查看当前已创建好的加密空间：

```
hdfs crypto -listZones
```

下面举几个官方的使用例子：

```
# 以普通用户的身份创建一个加密 key
hadoop key create myKey
```

```
# 以超级用户的身份创建一个空目录，并使之成为加密空间
hadoop fs -mkdir /zone
hdfs crypto -createZone -keyName myKey -path /zone
```

```
# 修改此目录权限为普通用户权限
hadoop fs -chown myuser:myuser /zone
```

```
# 以普通用户的身份进行 put 上传文件和 cat 查看文件操作
hadoop fs -put helloWorld /zone
hadoop fs -cat /zone/helloWorld
```

3.4 HDFS 纠删码技术

HDFS 纠删码技术指的是纠删码算法在 HDFS 中的实现，简称 HDFS EC。纠删码技术是一种用于数据恢复的技术。它对于 HDFS 带来的最大改变是可以使得 HDFS 不再依靠多副本机制来达到容错的效果。多副本机制的一个很大的弊端在于它对于存储空间的极大浪费，HDFS 纠删码技术的引入将会极大地改善这个问题。HDFS 纠删码功能将发布于 Hadoop 3.0 版本，如图 3-11 所示。

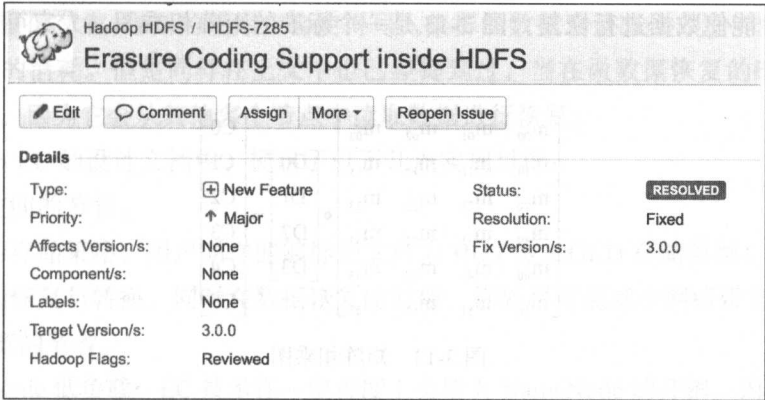


图 3-11 HDFS EC 相关 JIRA

在本节中，我们首先介绍纠删码技术本身的概念以及优劣势。随后介绍 HDFS 的纠删码技术，学习了解纠删码技术在 HDFS 中的实现原理。本节内容部分引用自 Apache 社区 JIRA 上 EC 的设计文档。

3.4.1 纠删码概念

纠删码技术是一种数据恢复技术，最早用于通信行业中数据传输中的数据恢复，是一种编码容错技术。它通过在原始数据中加入新的校验数据，使得各个部分的数据产生关联性。在一定范围内的数据出错情况下，通过纠删码技术都可以进行恢复。下面结合图片进行简单演示，首先有原始数据 n 个，加入 m 个校验数据块，如图 3-12 所示。

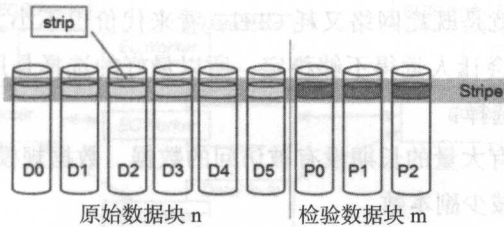


图 3-12 EC 数据块图

Parity 部分就是校验数据块，我们把一行数据块组称为条带 (strip)，每行条带由 n 个数据块和 m 个校验块组成。原始数据块和校验数据块都可以通过现有的数据块进行恢复，规则如下：

- 如果校验数据块发生错误，通过对原始数据块进行编码重新生成。
- 如果原始数据块发生错误，通过校验数据块的解码可以重新生成。

而且 m 和 n 的值并不是固定不变的，可以进行相应调整。可能有人会好奇，这其中到底是什么原理呢？其实道理很简单，我们把上面的图片看成矩阵，由于矩阵的运算具有可逆性，所以就能使数据进行恢复，图 3-13 是一个标准的矩阵相乘图，大家可以将二者进行关联。

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \\ m_{40} & m_{41} & m_{42} & m_{43} \\ m_{50} & m_{51} & m_{52} & m_{53} \end{bmatrix} \cdot \begin{bmatrix} D0 \\ D1 \\ D2 \\ D3 \end{bmatrix} = \begin{bmatrix} C0 \\ C1 \\ C2 \\ C3 \\ C4 \\ C5 \end{bmatrix}$$

图 3-13 矩阵相乘图

至于里面涉及的数学推理，大家可以自行寻找资料进行学习。

3.4.2 纠删码技术的优劣势

优势：纠删码技术作为一门数据恢复技术，有它自身的优势。它可以解决目前分布式系统、云计算中采用多副本机制来防止数据丢失的问题。副本机制确实可以解决数据丢失的问题，但是会使存储空间翻倍并很快耗光，这一点是非常致命的。EC 技术的运用就可以用来解决这个问题。

劣势：EC 技术的优势确实明显，但是它的使用也是需要付出一定代价的。一旦数据需要恢复，它会造成两大资源的消耗：

- 网络带宽的消耗，因为数据恢复需要去读其他的数据块和校验块。
- 进行编码、解码计算时需要消耗 CPU 资源。

概况来讲一句话，就是既耗网络又耗 CPU，看来代价也不小。所以这么来看，将此技术数用于线上服务可能会让人觉得不够稳定，所以最好的选择是用于冷数据的存储。以下两点原因可以支持这种选择：

- 冷数据集群往往有大量的长期没有被访问的数据，数据规模确实会比较大，采用 EC 技术，可以大大减少副本数。
- 冷数据集群基本稳定，耗资源量少，所以一旦进行数据恢复，也将不会对集群造成

大的影响。

出于上述两种原因，冷数据的存储无疑是一个很好的选择。

3.4.3 Hadoop 纠删码概述

纠删码技术在 Hadoop 中有两大优势：

- ❑ 存储空间的节省。EC 技术的单副本可以为集群节省多副本机制造成的额外存储空间的使用。这也是 EC 技术被运用到 HDFS 中的一个很重要的原因。
- ❑ 带宽流量的节省。EC 的使用，会使得写入集群的数据总量减少，进一步为集群节省了带宽的消耗。但是同样在上文中也已经提到过，当在做数据恢复的时候会比较耗费带宽，因为它会从其他多个节点中读取数据进行恢复。

在 Hadoop EC 的设计文档中，提出了以下几点实现目标：

- 1) 存储空间的节省。
- 2) 灵活的存储策略，用户同样能够标记文件为 HOT 或 COLD 存储类型。
- 3) 快速的恢复与转换，同时在数据恢复的时候，需要尽可能减少网络带宽的使用。
- 4) IO 带宽的节省。
- 5) NameNode 低负载。EC 技术在一定程度上会增大 NameNode 的开销。因为 NameNode 需要额外跟踪校验块的信息。在 EC 的实现过程中，需要尽可能降低 NameNode 的负载。
- 6) 对于用户的透明性、兼容性。EC 在 Hadoop 中的实现细节，需要保证对用户的绝对透明。而且用户可以在 EC 策略下正常使用其他功能，例如 HDFS 快照、HDFS 缓存、加密空间等等。

这里还有一点需要注意，EC 在 Hadoop 中的实现会直接改变原来 HDFS 默认的单副本策略，而副本数的减少会对 MR 任务的数据本地性造成一定影响。

图 3-14 是 Hadoop EC 总体架构图。

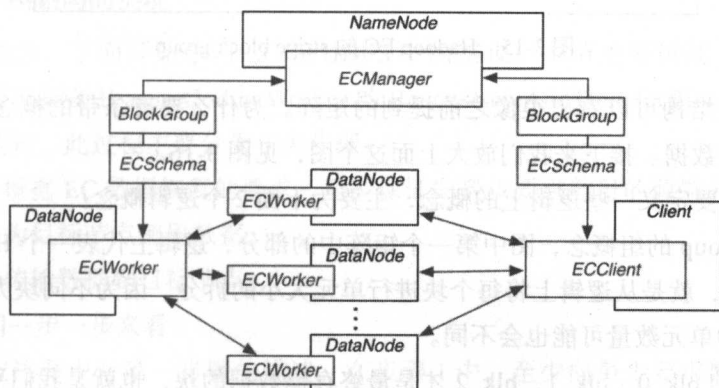


图 3-14 Hadoop EC 架构设计图

仔细观察上面的设计图，Hadoop EC 同样采用了 master/slave 的主从结构，在 NameNode、DataNode、Client 端都有相应的服务和角色。

3.4.4 纠删码技术在 Hadoop 中的实现

前面花了大量的篇幅介绍 EC 技术，相信大家已经或多或少了解了这项技术。现在才是本节的一个重点：Hadoop Erasure Coding 的实现。因为我们都知道，Hadoop 作为一个成熟的分布式系统，用的是三副本备份的策略，所以这项技术的引入对于 Hadoop 本身来说，意义是非常重大的。考虑到 EC 技术在 Hadoop 中的实现细节会比较复杂，所以这里不会逐行代码地进行分析，主要从大的方向上理一理它的实现思路。

1. EC 概念在 Hadoop 中的演变

EC 概念指的是数据块（data block）、校验块（parity block）、条带（stripe）等这些概念。那么这些概念在 HDFS 中是如何进行转化的呢？因为要想实现 EC 技术，至少在概念上相同的。HDFS 对这些概念的定义如下：

- 数据块、校验块在 HDFS 中的体现就是普通的数据块。
- 条带的概念需要将每个块进行分裂，每个块由若干个相同大小的单元（cell）组成，每个条带由一行单元构成，相当于从所有的数据块和校验块抽取出了一行。

图 3-15 为 HDFS 中条带的组成。

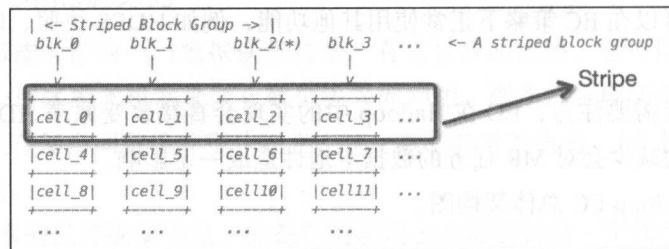


图 3-15 Hadoop EC 的 stripe block group

上面的横竖结构可以看出很像之前提到的矩阵。为什么要有条带的概念是因为矩阵运算会读到每行的数据。接下来我们放大上面这个图，见图 3-16。

在这里还需要定义一些逻辑上的概念，主要为下面 2 个逻辑概念：

- Block Group 的组概念，图中第一个矩阵中的部分，逻辑上代表一个 HDFS 文件。
- cell 概念，就是从逻辑上将每个块进行单元大小的拆分，因为不同块大小不同，所以不同块的单元数量可能也会不同。

图 3-16 中的 blk_0、blk_1、blk_2 才是最终存储数据的块，也就是我们平常说的 HDFS 中的块。

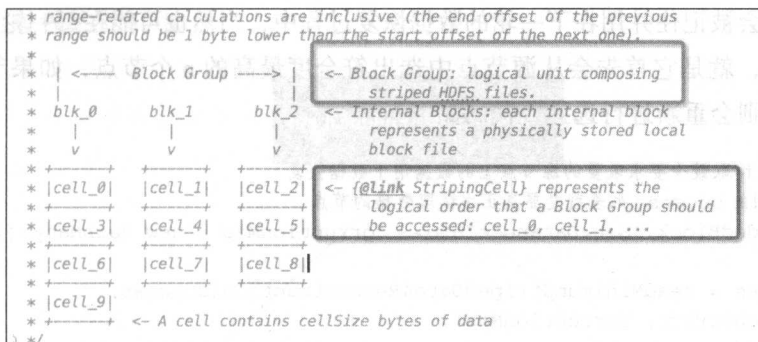


图 3-16 Hadoop EC 的 BlockGroup 详细结构图

条带的大小在 HDFS 中的计算逻辑如下：

```
final int stripeSize = cellSize * numDataBlocks;
```

stripeSize 就是一行的大小。获取块长度的实现逻辑如下：

```
// 计算每个条带的大小 (这里只计算数据块)
final int stripeSize = cellSize * numDataBlocks;
// 如果每个文件大小刚好整除条带大小, 则表明下面的所属块都是一样的大小
final int lastStripeDataLen = (int)(dataSize % stripeSize);
if (lastStripeDataLen == 0) {
    return dataSize / numDataBlocks;
}
// 否则根据每个块在 blockgroup 中的位置, 判断是否还要另外加上最后一个单元的大小
final int numStripes = (int)((dataSize - 1) / stripeSize + 1);
return (numStripes - 1L) * cellSize
    + lastCellSize(lastStripeDataLen, cellSize, numDataBlocks, i);
```

如果恰好最后一行条带长度为 0, 则说明每个块长度相等, 直接返回即可, 否则还要另外加上 lastCellSize 的大小。

2. HDFS 纠删码的实现

了解完概念, 下面开始学习 EC 在 HDFS 中的实现。本节主要讲述 EC 的数据恢复过程, 实现代码主要在 ErasureCodingWorker 的 ReconstructAndTransferBlock 类中。从此类的注释中可以获知, 此过程主要分为 3 大步骤:

步骤 1: 根据 EC 数据恢复的要求, 至少保证向最少要求数量的源端节点读取数据。

步骤 2: 为目标节点解码数据。

步骤 3: 传输数据到目标节点。

现在我们一步一步来看。

根据官方注释中对第一步骤的描述, 在步骤 1 中, 至少向最少要求数量的源端节点读取缓冲数据。如果部分源节点处于损坏状态或是数据落后的状态, 将会选择下一个源节点。

最好的源节点会被记住并用在下轮的数据恢复过程中，当然也可能在每一轮中被更新。

概况地说，就是它首先会从源节点中选出符合度最高的 n 个节点，如果节点中有坏的或是慢节点，则会重新进行选择，代码如下：

```
// 步骤 1: 读取最少要求数量的源节点上的数据用于数据恢复
// 这里返回的 success 列表就是要真正去读取数据的节点
Map<ExtendedBlock, Set<DataNodeInfo>> corruptionMap = new HashMap<>();
try {
    success = readMinimumStripedData4Reconstruction(success,
        toReconstruct, corruptionMap);
} finally {
    // 汇报坏块到 NameNode
    reportCorruptedBlocks(corruptionMap);
}
```

然后会对每个源节点新建相应的 StripedReader 对象以进行远程读取，远程读会用到 StripedReader 的 BlockReader 对象和 buffer 缓冲。

```
private StripedReader addStripedReader(int i, long offsetInBlock) {
    final ExtendedBlock block = getBlock(blockGroup, liveIndices[i]);
    // 创建 StripedReader 对象，用来读取远程数据
    StripedReader reader = new StripedReader(liveIndices[i], block, sources[i]);
    stripedReaders.add(reader);

    BlockReader blockReader = newBlockReader(block, offsetInBlock, sources[i]);
    if (blockReader != null) {
        initChecksumAndBufferSizeIfNeeded(blockReader);
        // StripedReader 实质读取数据的对象是 BlockReader
        reader.blockReader = blockReader;
    }
    reader.buffer = allocateBuffer(bufferSize);
    return reader;
}
```

StripedReader 结构如图 3-17 所示。

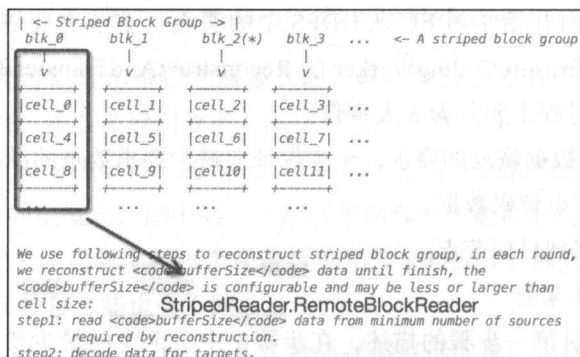


图 3-17 StripedReader 的 RemoteBlockReader

步骤 1 的总过程见图 3-18。

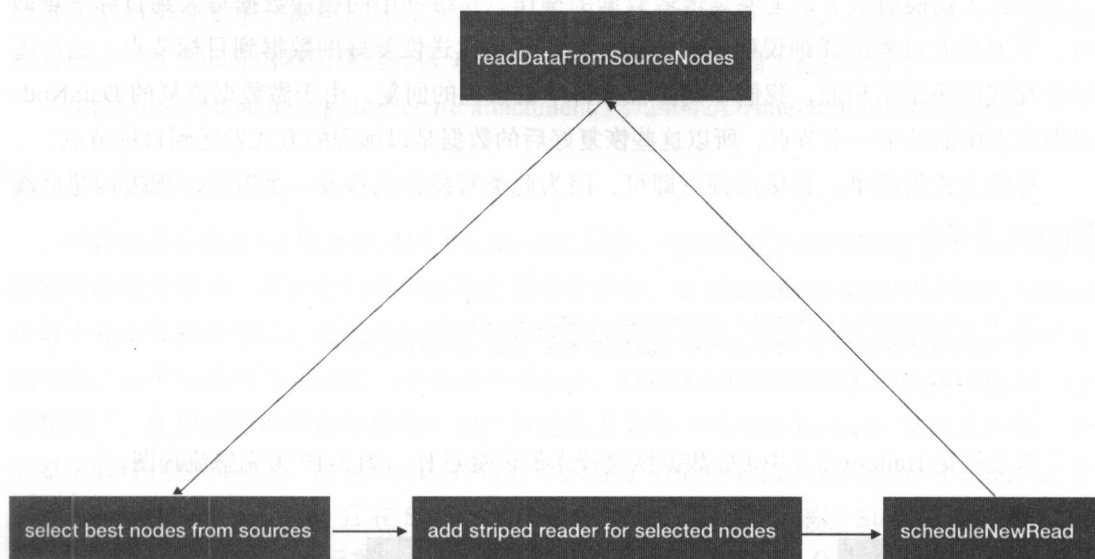


图 3-18 ErasureCodingWorker 的步骤 1 过程

在步骤 2 中，如果读取的源端块都是数据块类型的，我们需要调用编码操作。如果在源端块中存在一个校验块，则需要调用解码操作。注意在此过程中，只会读取数据一次，即能恢复所有条带块的数据。

步骤 2 主要点在于编解码数据的过程。步骤 1 已经把数据读到缓冲区了，步骤 2 就是计算的过程了。这里提到了很关键的一点：如果读取的源端块都是数据块类型的，需要调用编码操作。

编解码操作取决于恢复的对象，与之前提到的原则是一致的。相关代码如下：

// 步骤 2：为了目标块，进行解码恢复操作

```
reconstructTargets(success, targetsStatus, toReconstruct);
```

```
...
```

```
int[] erasedIndices = getErasedIndices(targetsStatus);
```

```
ByteBuffer[] outputs = new ByteBuffer[erasedIndices.length];
```

```
int m = 0;
```

```
for (int i = 0; i < targetBuffers.length; i++) {
```

```
    if (targetsStatus[i]) {
```

```
        targetBuffers[i].limit(toReconstructLen);
```

```
        outputs[m++] = targetBuffers[i];
```

```
    }
```

```
}
```

```
decoder.decode(inputs, erasedIndices, outputs);
```

```
...
```

这里使用的是解码的操作，说明恢复的块是数据块。

步骤 3 就很简单了，主要是传输数据的操作。将缓冲中的缓冲数据写入到目标节点即可。下面是此过程的详细说明：在步骤 3 中，直接发送恢复好的数据到目标节点。这与连续分配式的块副本相似，我们不需要检查每个数据包的回复。由于做数据恢复的 `DataNode` 是源节点中的其中一个节点，所以这些恢复好后的数据是以远程的方式发送到目标节点。

写的方式很简单，直接远程写即可，因为此类写操作只涉及一个节点，无需构建后续 Pipeline 的动作。

```
// 步骤 3: 传输数据到目标节点
if (transferData2Targets(targetsStatus) == 0) {
    String error = "Transfer failed for all targets.";
    throw new IOException(error);
}
```

以上就是 Hadoop 3.0 中 EC 数据恢复技术的恢复过程。图 3-19 为完整流程图。

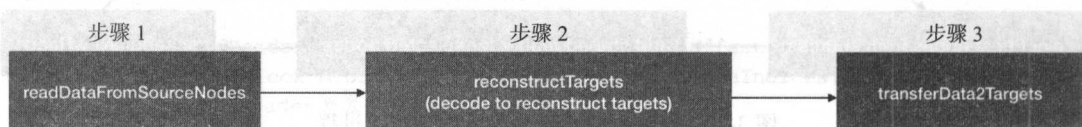


图 3-19 Hadoop EC 数据恢复步骤

3. 改进优化点

在官方注释中提到了 2 个优化点，在后续应该会被完善。

- ❑ 目前的数据没有采用本地读的方式，一律用远程方式进行数据读取。
- ❑ 目标数据的恢复传输并不返回数据包的确认码，不像 Pipeline 那样有很健全的一套体系。

HDFS EC 将会在 Hadoop 3.0 版本中发布，目前社区已经发布了 3.0 alpha 版本，大家可以提前试用此功能。

3.5 HDFS 对象存储：Ozone

对象存储是目前比较主流的服务存储形式，举两个比较典型的例子：亚马逊的 S3 存储和阿里云的 OSS 存储。对象存储指的是目标数据从对象中进行读写，然后通过键值获取对应的对象。整个存储的形式为 `key-object` 的存储方式，这样的好处在于方便用户的使用，不需要走复杂的操作流程。本节所要讲述的则是 HDFS 中的对象存储，在 HDFS 中我们也可以做到基于对象的存储。目前此功能正在开发中，名叫 Ozone，相关社区 JIRA：HDFS-7240 (Object store in HDFS)。在本节中，我们偏向于介绍 HDFS Ozone 的理论设计，因为

它在很多层面与现有 HDFS 的存储方式存在着比较大的差异。在理论设计方面，我们将主要讲述它的实现要求、高层设计以及部分细节的设计。在本节末尾，还会给出一个实际使用的例子。本节部分内容引用自社区 JIRA 上 Ozone 的设计文档，链接如下：

<https://issues.apache.org/jira/secure/attachment/12724042/Ozone-architecture-v1.pdf>。

3.5.1 Ozone 介绍

不管在亚马逊的 S3 服务还是阿里云的 OSS 服务，它们提供的服务都是基于 key-object 键值对的服务形式。而且它们很多的概念都比较类似，比如 bucket、object 的定义。object 是最小粒度级别的单位，代表的是最终存储数据的那个对象。而 bucket 则是其上一级的组织对象。一个 bucket 下可以有一个或多个 object。但是在 bucket 上一级的结构，就不一定都相同了，比如说你可以设计成每个用户只能拥有其唯一对应的 bucket，又或者声明一个对象来管理这些 bucket。而在 Ozone 的设计中采用了后者的做法。在 Ozone 中，bucket 存在于 StorageVolume 中，并且在 StorageVolume 中拥有唯一的名称。StorageVolume 会对其所包含的 bucket 对象进行数量上的配额限制。借此管理员可以分配许多有配额限制的 StorageVolume 给不同的用户。所以在 Ozone 中，每个用户直接对应的是 StorageVolume 而不是一堆的 bucket 列表。其中 bucket 等概念的命名规则如下：

一个 bucket 被两部分名字组合所唯一标识：storageVolumeName/bucketName。一个 object 的名称则被 storageVolumeName/bucketName/objectKey 三部分所标识。

Ozone 的数据组织结构如图 3-20 所示。

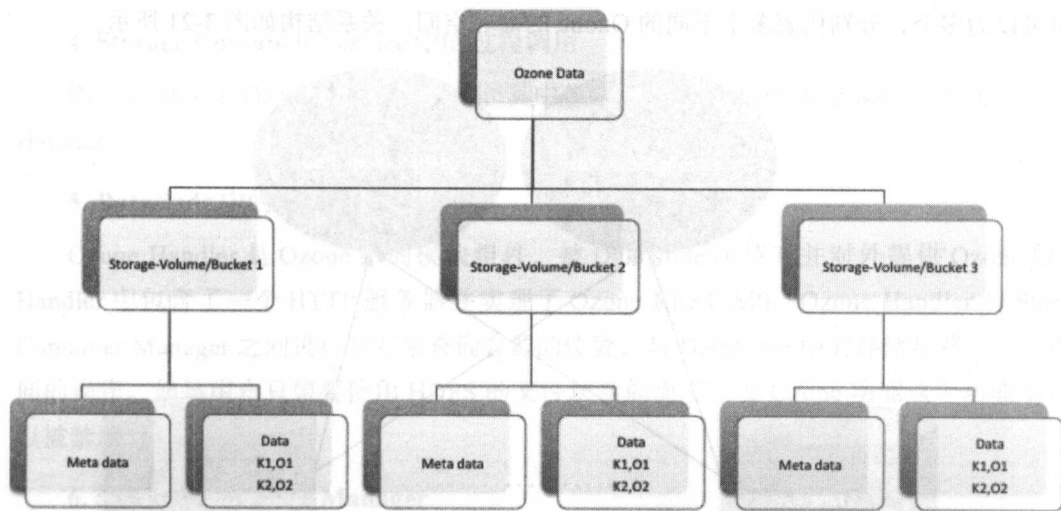


图 3-20 Ozone 的数据组织结构

在 HDFS Ozone 的设计中, 需要满足以下一些基本要求点:

- ❑ 管理员创建 StorageVolume。
- ❑ 创建 / 删除 bucket。每个 bucket 拥有一个独立的 URL, bucket 不能被重命名。只有 StorageVolume 的所属者或所属组才能创建 / 删除 volume 中的 bucket。
- ❑ 在一个 StorageVolume 中列出 bucket 列表。
- ❑ 根据给定的 key 在 bucket 中创建 / 删除 object 对象。对象的数据或值可以流式地传输到 Ozone 服务中。当对象被写满的时候将只会允许读操作, 同时不保证对象的局部写。
- ❑ 列举出 bucket 的内容。
- ❑ 创建 / 更新 / 删除 bucket 的 ACL 访问控制列表。

我们可以看到, 上面的一些基本要求在实际应用场景中还是很常用的。其中对于 bucket、object 这些概念具体的设计要求, 大家可以详细阅读前文中提到的 Ozone 的设计文档链接, 在此不再详细介绍。

3.5.2 Ozone 的高层级设计

1. 与 HDFS 共享 DataNode 数据存储

Ozone 的出现使得 HDFS 在使用方式上将会与原来块数据读写的方式有很大不同, 所以在这里我们将会以一个独立的 block pool 来存储 Ozone 上的数据。也就是说 DataNode 会同时为 HDFS 的 block pool 和 Ozone 的 block pool 存储数据。同样的, Ozone 的 block pool 也可以为多个, 分别代表多个不同的 Ozone 的命名空间。关系结构如图 3-21 所示。

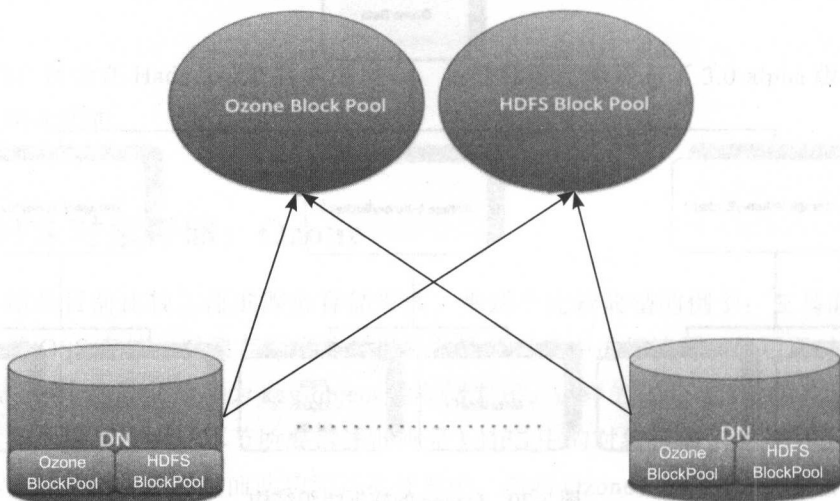


图 3-21 HDFS 和 Ozone 的 BlockPool 结构

2. 存储容器 (Storage Container)

一个存储容器从概念上来说,指的是用来存储 Ozone 数据(其实也是 bucket 中的数据)和 Ozone 元数据的一个存储单元。存储容器与 HDFS 的块一样,共同存储于 DataNode 上。但是与 HDFS 不同的一点, Ozone 没有一个类似于 NameNode 这样的中心控制节点。相反它是一个分离的元数据的存储与管理器,这些元数据分布式地存在于各个存储容器中。

一个 bucket 可以拥有百万数量级的对象并且在存储的大小级别上可以达到 TB 级别,这远远大于一个存储容器所能承受的(此处可以类比于块)。因此一个 bucket 会被分为很多分区(partition),每片分区会存储在一个存储容器中。(一个存储容器可以包含最大值数量的分区,然而对象只能来自一个 bucket)在目前的初始设计实现中,为了实现的简易性,一个 object 对象完全存在于单一存储容器中。

3. 存储容器标识符

每个存储容器会被独立的 storage container 标识符所独立标识,它是一个逻辑上的独立标识(类似于 HDFS 中的块 Id)。

对象的 key 会映射到 storage container 标识符。这个标识符会传入 Storage Container Manager (存储容器管理器)中来定位包含目标对象容器所在的 DataNode。在此,Storage Contain Manager 可以完全理解为 BlockManager 的角色。类似的, bucket 的名称也会映射到 storage container 标识符,这个容器保存有 bucket 的元数据。在后面的小节中,我们会具体提到这个映射关系是如何实现的。在存储容器相关的设计实现中,要满足尽可能复用已有代码的原则。

4. Storage Container Service 中的过程调用

图 3-22 展示了 Ozone 中的过程调用,其中包含了 Storage Container Service 服务和 Ozone Handler。

5. DataNode 中的 Ozone Handler

Ozone Handler 是 Ozone 中的模块组件,被 DataNode 所持有并对外提供 Ozone 服务。Handler 中包含了一个 HTTP 服务器并实现了 Ozone REST API。Ozone Handler 与 Storage Container Manager 之间进行交互来查询容器的位置,与 DataNode 中的存储容器交互实现不同的操作。如果用户只想要使用 HDFS 的文件块功能而不需要 Ozone 功能这个功能组件可以被禁用。

6. Storage Container Manager

Storage Container Manager 非常类似于 HDFS 中 BlockManager 的角色。Storage Container Manager 从各个 DataNode 中收集心跳,处理存储容器的报告并跟踪每个存储容器的位

置。它在内部维护了一个存储容器映射图，用前缀匹配的方式来查询存储容器。图 3-23 为 Storage Container Manager 与 DataNode 的交互图。

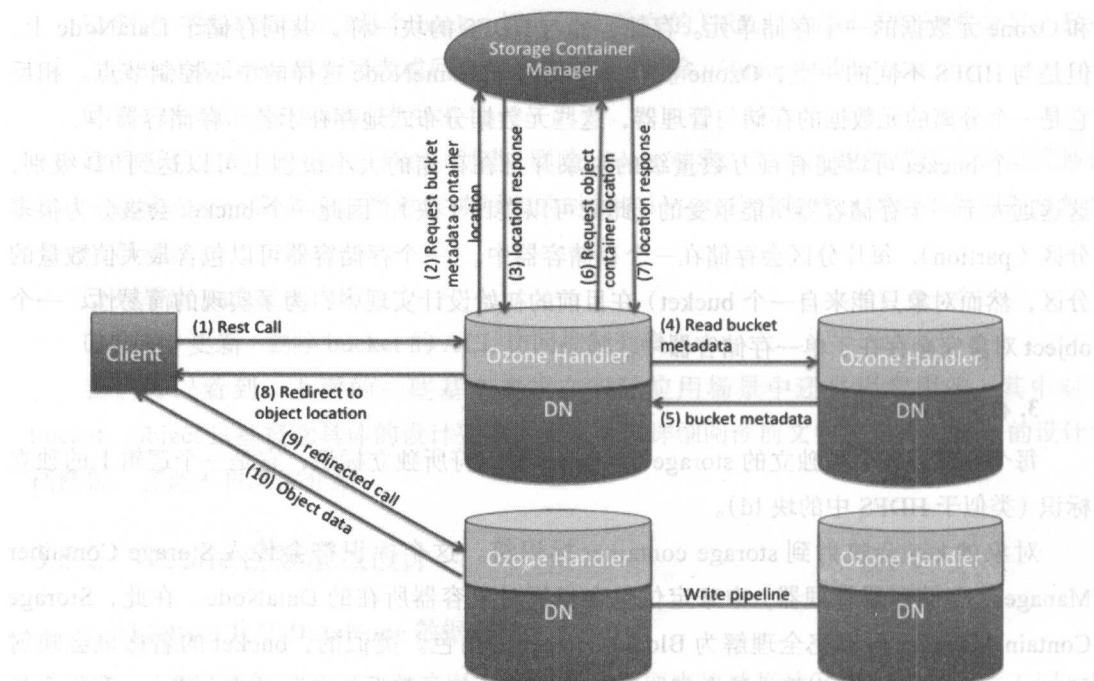


图 3-22 Storage Container Service 过程调用

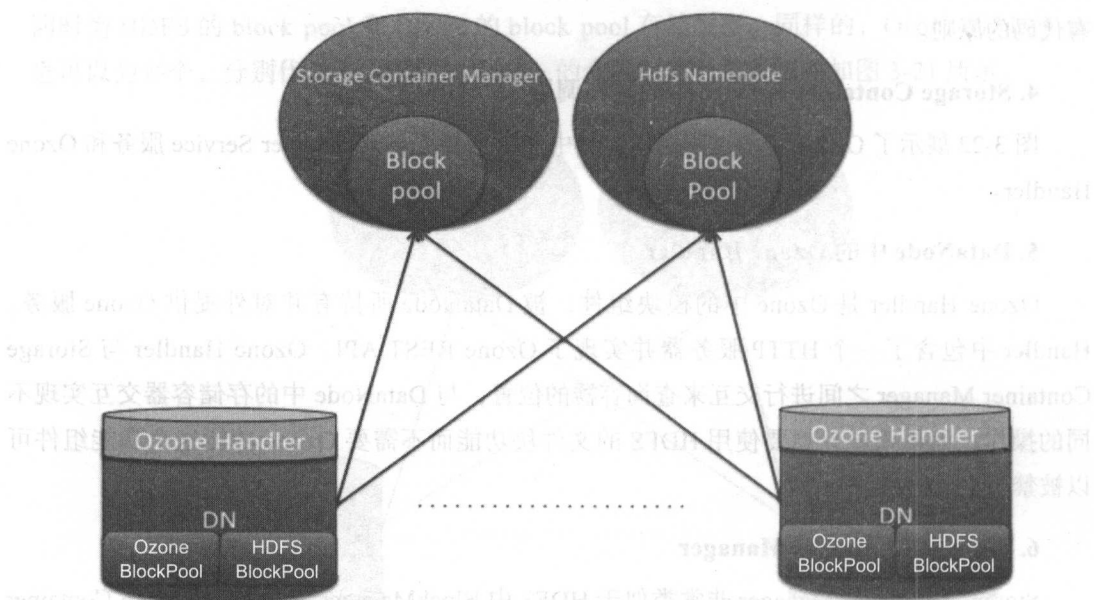


图 3-23 Storage Container Manager 与 DataNode 的交互

3.5.3 Ozone 的实现细节

1. 映射 object-key 到存储容器

下面再来看看其中的一些细节实现。其中大家比较关心的一点在于 key-object 到存储容器的映射。因为对于用户而言，他所面对的就是一个 key，那么我们如何找到它对应存储在存储容器中的 object 的数据呢？在 Ozone 中，采用哈希分区的方式来映射 key 到它所存储的存储容器。当容器逐渐变大，我们会对此进行分割，并用扩展哈希算法重新映射 key 到对应的容器。同时在 Storage Container Manager 中更新此映射关系。

2. 存储容器的实现

在前文中我们已经提到过，存储容器完全类比于 HDFS 中的块，这样可以尽可能地复用 HDFS 现有的 block pool 管理方面的功能代码。因此我们会将存储容器作为 block 类的扩展类来实现。一个 HDFS 块由一个标识符、生成时间记录和大小组成。这三个属性同样可以应用到存储容器中。

而在数据的一致性上，我们需要为存储容器实现一个新的 Pipeline 机制。因为它将要求一套新的错误恢复机制。

3.5.4 Ozone 的使用

由于目前 HDFS Ozone 的开发工作是基于分支 HDFS-7240 的，还没合入 trunk，所以我们获取 hadoop-trunk 的代码后得先切换一下分支。学习一个新的功能特性最快的方法是从看懂它的单元测试开始，笔者挑选了下面一个典型的例子，在这个例子中，我们将会看到 volume、bucket、object 的数据是如何运用的。

```
// Ozone 的使用方法测试
@Test
public void testPutAndGetMultiChunkKey() throws Exception {
    // 定义一个待添加的 volume 的名称
    String volumeName = nextId("volume");
    // 定义一个待添加的 bucket 的名称
    String bucketName = nextId("bucket");
    // 定义一个待创建的 key 的名称
    String keyName = nextId("key");
    // 定义待写入此 key 的对象的数据大小
    int keyDataLen = 3 * CHUNK_SIZE;
    // 创建待写入的数据
    String keyData = buildKeyData(keyDataLen);
    // 在 Ozone 中创建一个 volume
    OzoneVolume volume = ozoneClient.createVolume(volumeName, "bilbo", "100TB");
    assertNotNull(volume);
    // 获取此 volume，并检验此 volume 的信息是否为我们所期望的
```

```

assertEquals(volumeName, volume.getVolumeName());
assertEquals(ozoneClient.getUserAuth(), volume.getCreatedBy());
assertEquals("bilbo", volume.getOwnerName());
assertNotNull(volume.getQuota());
// 比较 Volume 的限制值是否也是对的
assertEquals(OzoneQuota.parseQuota("100TB").sizeInBytes(),
    volume.getQuota().sizeInBytes());

// 在此 volume 下新建 bucket
OzoneBucket bucket = volume.createBucket(bucketName);
// 检测此 bucket 不为空
assertNotNull(bucket);
// 检测此 bucket 是我们所期望创建的名称
assertEquals(bucketName, bucket.getBucketName());

// 在此 bucket 下存入 key 以及 key 所对应的 object 对象数据
bucket.putKey(keyName, keyData);
// 传入 key, 进行数据的获取, 判断数据是否一致
assertEquals(keyData, bucket.getKey(keyName));
}

```

通过上面的例子我们可以看出, Ozone 使用的方式非常简单, 根本无须写任何关于 DFSClient 端读写文件数据的代码。但是同样需要指出的是, 由于 Ozone 的使用方式完全独立于现有的 NameNode、DataNode 之间块数据管理的形式, 因此 HDFS 目前主要的一些功能特性可能无法运用在 Ozone 上。Ozone 的出现应该说丰富了大家使用 HDFS 的方式, 至少如果我们想做对象存储的话, 就不用单独搞另外一套系统来实现了。

3.6 小结

本章主要讲述了 HDFS 内部一些用户熟悉度较低的功能特性。在这些特性中, ViewFs 和 WebHDFS 比较实用, 大家只需理解其原理并懂得如何使用即可。而本章的难点在于剩下的三节: HDFS EC、Encryption zone 和 Ozone。这三节涉及一些数学的算法以及抽象的设计思想, 对于这三节的学习, 需要读者反复阅读与理解。

第二部分 Part 2

细节实现篇

HDFS 的块处理

本章主要介绍 HDFS 存储数据的最小单元：块。了解块处理过程对于排除问题具有十分重要的意义。本章分为三个方面。首先介绍块检查命令 `fsck`，此节将会告诉我们块丢失了怎么查。其次介绍 HDFS 中多余块的删除方法。最后介绍块的上报处理过程，了解块在 NameNode 中的周转过程。

4.1 HDFS 块检查命令 `fsck`

在 HDFS 中，所有的文件都是以块的形式存在的。集群在运行过程中难免会发生块损坏的情况，这个时候就可以使用 HDFS 的块检查命令：`fsck`。在这个工具命令下，不仅仅有用于检测坏块的子命令，还有相应的处理命令，在很多时候可以帮助我们轻松地清理集群中的损坏文件。本节将要介绍 `fsck` 命令的使用场景、`fsck` 命令各个参数的使用 and `fsck` 块检查命令的调用过程。

其实对于 `fsck` 命令本身，熟悉 Linux 操作系统的人，可能或多或少听过或使用过。`fsck` 命令的全称是 `file system check`，意为文件系统检测命令，此命令可以作为一种修复命令。本节不会介绍操作系统的 `fsck` 怎么用，主要讲述 HDFS 下 `fsck` 的使用，`bin/hdfs fsck` 命令下包含有很多可选参数。

4.1.1 `fsck` 参数使用

笔者在测试集群中输入 `hdfs fsck` 命令后，获取了如下帮助信息，在此信息中展示了最

全的参数使用说明：

```
$ hdfs fsck
```

```
Usage: hdfs fsck <path> [-list-corruptfileblocks | [-move | -delete |  
-openforwrite] [-files [-blocks [-locations | -racks]]]]
```

- <path> 目标扫描的路径名称
- -move 移动损坏的文件到 /lost+found 目录下
- -delete 删除损坏的文件
- -files 输出被检测到的文件
- -openforwrite 输出正在被写的文件
- -includeSnapshots 如果检测的路径下包含了快照目录的话，输出快照信息
- -list-corruptfileblocks 输出损坏的块信息
- -blocks 输出块的报告信息
- -locations 输出每个块的位置信息
- -racks 输出块所属节点的机架信息
- -storagepolicies 输出块上设置的存储策略信息
- -blockId 输出指定块 Id 对应的块信息、所在机架信息等等

简单对上面的命令进行总结，首先是必填参数和命名：

```
bin/hdfs fsck <path>
```

然后是以下可选参数：-move、-delete、-files、-openforwrite、-includeSnapshots、-list-corruptfileblocks、-blocks、-locations、-racks、-storagepolicies、-blockId。

具体参数功能对应的逻辑代码会在下文的分析中详细地讲述。

4.1.2 fsck 过程调用

fsck 过程的调用指的是从终端窗口输入到最终 fsck 在 HDFS 内部执行完毕的整个过程。中间经过的类其实不多，图 4-1 为 fsck 执行过程。



图 4-1 fsck 过程调用图

上图的调用形式可以说是三层调用结构。DFSck 是暴露在最外层的类，下面再来梳理一下中间的过程：

1) 输入 `fsck <path>` 直接调用到的是类 `DFSck`。`DFSck` 内部会以 `http` 请求的方式, 根据参数构造 `URL` 请求地址, 发送请求到下一个处理对象中。

2) 下一个处理对象是 `FsckServlet`。`FsckServlet` 相当于一个过渡者的角色, 用于马上调用真正操作类 `NamenodeFsck`。

3) `NamenodeFsck` 会取出请求参数, 然后在 `HDFS` 内部做真正的 `fsck` 检测操作。

4.1.3 fsck 原理分析

`fsck` 原理分析将会展示更加细致的 `fsck` 过程调用。根据上文提到的三层调用, 我们将此过程分为三个分析部分。

1. DFSck 请求构造

我们可以把此类比成 `DFSAdmin`。首先介绍命令输入处理入口方法:

```
public int run(final String[] args) throws IOException {
    if (args.length == 0) {
        printUsage(System.err);
        return -1;
    }

    try {
        return UserGroupInformation.getCurrentUser().doAs(
            new PrivilegedExceptionAction<Integer>() {
                @Override
                public Integer run() throws Exception {
                    // 传入输入的参数进行命令的执行
                    return doWork(args);
                }
            });
    } catch (InterruptedException e) {
        throw new IOException(e);
    }
}
```

在 `doWork` 方法中, 进行了参数的判别分类, 同时开始构造不同的参数请求。

```
private int doWork(final String[] args) throws IOException {
    final StringBuilder url = new StringBuilder();

    url.append("/fsck?ugi=").append(ugi.getShortUserName());
    String dir = null;

    boolean doListCorruptFileBlocks = false;
    // 遍历参数, 进行相关参数的请求设置
    for (int idx = 0; idx < args.length; idx++) {
        if (args[idx].equals("-move")) { url.append("&move=1"); }
        else if (args[idx].equals("-delete")) { url.append("&delete=1"); }
    }
}
```



```

else if (args[idx].equals("-files")) { url.append("&files=1"); }
else if (args[idx].equals("-openforwrite")) { url.append("&openforwrite=1"); }
else if (args[idx].equals("-blocks")) { url.append("&blocks=1"); }
else if (args[idx].equals("-locations")) { url.append("&locations=1"); }
else if (args[idx].equals("-racks")) { url.append("&racks=1"); }
else if (args[idx].equals("-storagepolicies")) { url.append
("&storagepolicies=1"); }
...

```

不同类型的参数后面接的参数值不一定相同，比如 `-blockId` 后面会跟连续的块 Id。

```

...
} else if (args[idx].equals("-blockId")) {
    StringBuilder sb = new StringBuilder();
    idx++;
    while(idx < args.length && !args[idx].startsWith("-")){
        sb.append(args[idx]);
        sb.append(" ");
        idx++;
    }
    url.append("&blockId=").append(URLEncoder.encode(sb.toString(), "UTF-8"));
...

```

请求 url 构造好之后，发起请求：

```

URL path = new URL(url.toString());
URLConnection connection;
try {
    connection = connectionFactory.openConnection(path, isSpnegoEnabled);
} catch (AuthenticationException e) {
    throw new IOException(e);
}

```

随后获取响应回复，直接输出到终端上。

```

InputStream stream = connection.getInputStream();
BufferedReader input = new BufferedReader(new InputStreamReader(stream, "UTF-8"));
String line = null;
String lastLine = null;
int errCode = -1;
try {
    while ((line = input.readLine()) != null) {
        out.println(line);
        lastLine = line;
    }
} finally {
    input.close();
}

```

到此，DFSck 最外层的调用过程就走通了。

2. FsckServlet 请求处理

上个步骤中 url 请求会转到 FsckServlet 中处理, FsckServlet 类似代理人的角色, 紧接着调用 NamenodeFsck 进行处理:

```
// doGet 处理 fsck 的请求
@Override
public void doGet(HttpServletRequest request, HttpServletResponse response
    ) throws IOException {
    @SuppressWarnings("unchecked")
    final Map<String,String[]> pmap = request.getParameterMap();
    ...

    final UserGroupInformation ugi = getUGI(request, conf);
    try {
        ugi.doAs(new PrivilegedExceptionAction<Object>() {
            @Override
            public Object run() throws Exception {
                NameNode nn = NameNodeHttpServer.getNameNodeFromContext(context);

                final FSNamesystem namesystem = nn.getNamesystem();
                final BlockManager bm = namesystem.getBlockManager();
                final int totalDatanodes =
                    namesystem.getNumberOfDatanodes(DatanodeReportType.LIVE);
                // 调用 NamenodeFsck 的 fsck 方法进行处理
                new NamenodeFsck(conf, nn,
                    bm.getDatanodeManager().getNetworkTopology(), pmap, out,
                    totalDatanodes, remoteAddress).fsck();

                return null;
            }
        });
    } catch (InterruptedException e) {
        response.sendError(400, e.getMessage());
    }
}
```

3. NamenodeFsck 的 fsck 处理

最后一个步骤调用的是 NamenodeFsck 的 fsck 方法。在进入这个方法之前, 先了解这个类内部的一些关键变量:

```
private String lostFound = null;
private boolean lfInitied = false;
private boolean lfInitiedOk = false;
// 显示文件标识
private boolean showFiles = false;
// 显示已打开的文件标识
private boolean showOpenFiles = false;
```

```
// 显示块信息标识
private boolean showBlocks = false;
// 显示位置信息标识
private boolean showLocations = false;
// 显示机架信息标识
private boolean showRacks = false;
// 显示存储策略标识
private boolean showStoragePolicies = false;
// 显示损坏文件标识
private boolean showCorruptFileBlocks = false;
```

这些布尔类型的变量对应的就是 fsck 帮助信息中所展示的各个参数。fsck 方法内部的处理顺序看起来有点乱，为了便于大家理解，这里对指定参数进行指定分析。

第一个参数方法 -list-corruptfileblocks，展示丢失 / 损坏的块。

```
if (showCorruptFileBlocks) {
    listCorruptFileBlocks();
    return;
}
```

调用到同名方法 listCorruptFileBlocks。

```
private void listCorruptFileBlocks() throws IOException {
    Collection<FSNamesystem.CorruptFileBlockInfo> corruptFiles = namenode.
        getNameSystem().listCorruptFileBlocks(path, currentCookie);
    int numCorruptFiles = corruptFiles.size();
    ...
    out.println("Cookie:\t" + currentCookie[0]);
    for (FSNamesystem.CorruptFileBlockInfo c : corruptFiles) {
        out.println(c.toString());
    }
    out.println("\n\nThe filesystem under path '" + path + "' has " + filler
        + " CORRUPT files");
    out.println();
}
```

此方法最终会调用到 FSNamesystem 的 listCorruptFileBlocks 方法，注意这里还传入了一个特别的参数 currentCookie，这个参数的作用比较巧妙。进入 FSNamesystem 的方法，首先初始化损坏文件块对象：

```
ArrayList<CorruptFileBlockInfo> corruptFiles = new ArrayList<CorruptFileBlockInfo>();
```

方法返回的对象即为损坏文件块对象。

然后进入关键的损坏文件的判断逻辑：

```
// 做一个快速的检查，如果当前没有损坏的文件，无须申请锁操作
if (blockManager.getMissingBlocksCount() == 0) {
    if (cookieTab[0] == null) {
        cookieTab[0] = String.valueOf(getIntCookie(cookieTab[0]));
```

```

    }
    if (LOG.isDebugEnabled()) {
        LOG.debug("there are no corrupt file blocks.");
    }
    return corruptFiles;
}

```

blockManager 的 **getMissingBlocksCount** 方法取的就是损坏块队列的大小:

```

public long getMissingBlocksCount() {
    // 此处实际返回的是损坏块的数量
    return this.neededReplications.getCorruptBlockSize();
}

```

如果此方法的 **Count** 返回值有值, 即大于 0, 则方法执行继续:

```

// 获取损坏块的块迭代器
final Iterator<Block> blkIterator = blockManager.getCorruptReplicaBlockIterator();
// 取出 cookie 值作为标记位, 跳过标记下标之前的文件, 代表已经浏览过
int skip = getIntCookie(cookieTab[0]);
for (int i = 0; i < skip && blkIterator.hasNext(); i++) {
    blkIterator.next();
}

while (blkIterator.hasNext()) {
    Block blk = blkIterator.next();
    final INode inode = (INode)blockManager.getBlockCollection(blk);
    // 更新 skip 跳过值
    skip++;
    if (inode != null && blockManager.countNodes(blk).liveReplicas() == 0) {
        String src = FSDirectory.getFullPathName(inode);
        if (src.startsWith(path)){
            // 将损坏块文件信息加入 corruptFiles 列表内
            corruptFiles.add(new CorruptFileBlockInfo(src, blk));
            count++;
            if (count >= DEFAULT_MAX_CORRUPT_FILEBLOCKS_RETURNED)
                break;
        }
    }
}
// 更新 cookie 标记值
cookieTab[0] = String.valueOf(skip);

```

cookie 的作用如上注释所说。获取到此返回损坏文件列表后, 会在上一方法中将结果输出:

```

for (FSNamesystem.CorruptFileBlockInfo c : corruptFiles)
{
    out.println(c.toString());
}

```

4. fsck -path 默认处理方法

fsck 的默认处理方法指的是 fsck+path 的方法，为什么紧接着讲这个方法呢？因为 fsck 的 path 方法处理也包括了扫描损坏块的方法，但是在逻辑上与 -list-corruptfiles 竟然不一样，这一点笔者在阅读的时候，会感到比较奇怪。首先用户传入的 path 会被传入到内部方法 check 中处理：

```
Result res = new Result(conf);

check(path, file, res);

out.println(res);
out.println(" Number of data-nodes:\t\t" + totalDatanodes);
out.println(" Number of racks:\t\t" + networkTopology.getNumOfRacks());
```

然后会进行目录、文件的判断，如果是目录，则进行递归调用：

```
if (file.isDir()) {
    // 如果快照目录包含此路径，则递归快照目录下的 path
    if (snapshottableDirs != null && snapshottableDirs.contains(path)) {
        String snapshotPath = (path.endsWith(Path.SEPARATOR) ? path : path
            + Path.SEPARATOR
            + HdfsConstants.DOT_SNAPSHOT_DIR;
        HdfsFileStatus snapshotFileInfo = namenode.getRpcServer().getFileInfo(
            snapshotPath);
        check(snapshotPath, snapshotFileInfo, res);
    }
    ...
    do {
        assert lastReturnedName != null;
        thisListing = namenode.getRpcServer().getListing(
            path, lastReturnedName, false);
        if (thisListing == null) {
            return;
        }
        HdfsFileStatus[] files = thisListing.getPartialListing();
        // 如果此 path 是目录的话，递归遍历此 path 的子文件
        for (int i = 0; i < files.length; i++) {
            check(path, files[i], res);
        }
        lastReturnedName = thisListing.getLastName();
    } while (thisListing.hasMore());
    return;
}
```

在接下来分析检测文件时，会进行相应指标统计值的更新。

```
isOpen = blocks.isUnderConstruction();
if (isOpen && !showOpenFiles) {
```

```

// 更新正在被写入文件的一些指标信息, 供不同 fsck 的参数使用
res.totalOpenFilesSize += fileLen;
res.totalOpenFilesBlocks += blocks.locatedBlockCount();
res.totalOpenFiles++;
return;
}
// 更新文件块相关的信息
res.totalFiles++;
res.totalSize += fileLen;
res.totalBlocks += blocks.locatedBlockCount();

```

下面是 fsck 默认的损坏块的判断逻辑:

```

...
for (LocatedBlock lBlk : blocks.getLocatedBlocks()) {
    ExtendedBlock block = lBlk.getBlock();
    boolean isCorrupt = lBlk.isCorrupt();
    String blkName = block.toString();
}
...

```

这里利用了 LocatedBlock 内部的 isCorrupt 的方法, 然后进行 corrupt 计数累加:

```

// 检查块是否损坏了
if (isCorrupt) {
    corrupt++;
    res.corruptBlocks++;
    out.print("\n" + path + ": CORRUPT blockpool " + block.getBlockPoolId() +
        " block " + block.getBlockName()+"\n");
}

```

在这里, 丢失块的判断逻辑是独立于损坏块的。

```

// 重新进行块副本数的统计
NumberReplicas numberReplicas =
    namenode.getNameSystem().getBlockManager().countNodes(block.
        getLocalBlock());
// 获取存在的副本数
int liveReplicas = numberReplicas.liveReplicas();

// 如果当前副本数为 0, 则表明是丢失块
if (liveReplicas == 0) {
    report.append(" MISSING!");
    res.addMissing(block.toString(), block.getNumBytes());
    missing++;
    missize += block.getNumBytes();
} else {

```

重新回顾以上 check 方法中的这两类块判断逻辑。对于第二个丢失块的判断逻辑, 笔者认为是没有问题的, 但是第一个损坏块的判断逻辑可能有点问题, 尽管说 LocatedBlock 提

供了内部方法 `isCorrupt`，但是在查询 `isCorrupt` 的调用处时发现绝大多数情况下都是 `false` 参数默认传入的，而且在数据实时性和有效性上，这个方法没有 `-list-corruptfiles` 参数来得快与准（笔者个人观点，可能理解不同）。因为 `-list-corruptfiles` 是直接从 `FSNamesystem` 类中获取的，从大的层面来说，代表的已经是最新损坏块数据的信息了。

5. fsck -delete/-move

这两个命令的作用是找到损坏块之后，做进一步处理，就是下面两行代码所控制的路径：

```
...
} else {
    // 如果输入了 -move 参数，则将块移至 lost+found 目录下
    if (doMove) copyBlocksToLostFound(parent, file, blocks);
    // 如果输入了 -delete 参数，则删除指定路径下损坏的文件块
    if (doDelete) deleteCorruptedFile(path);
}
...
```

`LostFound` 指的是 `/lost+found` 目录，`-move` 参数会将损坏块文件移至此目录下，而 `-delete` 则会调用直接删除文件的方法。

```
private void deleteCorruptedFile(String path) {
    try {
        // 调用 NameNode 删除文件的方法
        namenode.getRpcServer().delete(path, true);
        LOG.info("Fsck: deleted corrupt file " + path);
    } catch (Exception e) {
        LOG.error("Fsck: error deleting corrupted file " + path, e);
        internalError = true;
    }
}
```

这两个命令在清理损坏数据的场景中比较有用。如果集群中存在大量损坏块数据的情况，不及时进行清理，会出现大量客户端读写操作的失败，因为元数据虽然存在，但是真实数据已经损坏，读写操作必然会抛出异常。

6. fsck 辅助显示参数

以上几个是 `fsck` 的主要参数，下面是一些辅助的次要参数。

* `-locations/-racks`

```
if (showLocations || showRacks) {
    StringBuilder sb = new StringBuilder("");
    for (int j = 0; j < locs.length; j++) {
        if (j > 0) { sb.append(", "); }
```

```

// 如果指定输出机架信息, 则进行输出
if (showRacks)
    sb.append(NodeBase.getPath(locs[j]));
else
    sb.append(locs[j]);
}
sb.append(' ');
report.append(" " + sb.toString());
}

* -storagepolicies

    if (this.showStoragePolcies) {
        storageTypeSummary = new StoragePolicySummary(
            namenode.getNameSystem().getBlockManager().getStoragePolicies());
    }

    ...
    // 输出存储策略信息
    if (this.showStoragePolcies) {
        out.print(storageTypeSummary.toString());
    }

* -includeSnapshots
    此参数会获取 NameNode 快照中的目录信息

    if (snapshottableDirs != null) {
        SnapshottableDirectoryStatus[] snapshotDirs = namenode.getRpcServer()
            .getSnapshottableDirListing();
        if (snapshotDirs != null) {
            for (SnapshottableDirectoryStatus dir : snapshotDirs) {
                snapshottableDirs.add(dir.getFullPath().toString());
            }
        }
    }
}

```

在这些参数执行期间, 会伴随着结果的输出, 所以你将会看到路径的信息被展示到终端, 最后是总结报告的输出, 如下所示:

```

Total size:      88.13 KB
Total dirs:      14
Total files:     20
Total symlinks:  0
Total blocks (validated): 20 (avg. block size 4512 B)
*****
UNDER MIN REPL'D BLOCKS: 20 (100.0 %)
dfs.namenode.replication.min: 1
CORRUPT FILES: 20
MISSING BLOCKS: 20
MISSING SIZE:    88.13 KB

```



```

CORRUPT BLOCKS: 20
*****
Minimally replicated blocks: 0 (0.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 0
Average block replication: 0.0
Corrupt blocks: 20
Missing replicas: 0
Number of data-nodes: 0
Number of racks: 0
FSCK ended at Tue Mar 29 11:10:33 CST 2016 in 10 milliseconds

```

The filesystem under path '/' is CORRUPT

NamenodeFsck 的处理过程和参数控制如上内容所述,方法主要集中在 fsck 和 check 的两个方法内,并根据所选参数选择性地输出中间结果。图 4-2 为 fsck 的执行逻辑。

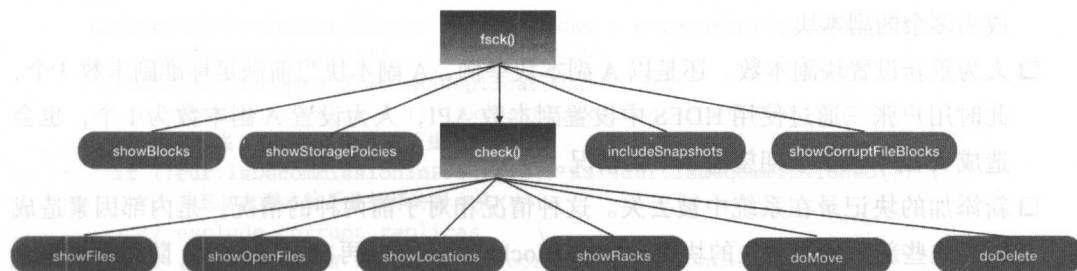


图 4-2 fsck 检测执行逻辑图

4.1.4 fsck 使用场景

fsck 块检查命令有许多使用场景,下面是两类比较常见的使用场景。

第一类场景,损坏块、丢失块的检查与处理。通过 fsck 命令附带目标检查路径参数能够得到详细的检测结果。如果检测出有丢失或损坏的块,通过 -delete 参数可以直接进行删除,防止丢失的文件块造成程序执行的异常。

第二类场景,块信息的查询。通过 fsck 的 -files、-bloks 等参数可以很详细地得到块存储位置等信息,甚至还能通过块 Id 查询该块对应的位置信息。

4.2 HDFS 如何检测并删除多余副本块

在 HDFS 中,每时每刻都在进行着大量块的创建和删除操作,从而使得集群中维护的

都是有效的数据，而在块删除的场景中，有一类情况比较特殊：删除多余的副本块。当一个文件块的现有副本数超过它的期望副本数，多出来的那些块即“多余的副本块”。多余副本块的出现往往发生在某些特殊的场合，所以本节要讲述的主要内容有：第一，HDFS 如何检测这些多余副本块；第二，检测出来了如何进行删除；第三，删除的逻辑策略。

4.2.1 多余副本块以及发生的场景

下面举例说明多余副本块：假设集群中有 3 个 A 副本块，满足标准的三副本策略，但是发生了某种操作后，A 副本块突然变为了 5 个，为了达到副本块的标准块数 3 个，系统会进行多余 2 块副本的清除动作，而这个清除动作就是本节所要重点讲述的。多余副本块的现象是比较好理解的，但是到底有哪些潜在的原因或条件会触发多余副本块的发生呢（在此指的是 HDFS 中）？笔者通过对 HDFS 源码的阅读，总结出以下 3 点：

- ❑ ReCommission 节点重新上线。这类操作是运维操作引起的，节点下线操作会导致大量此节点的块在集群中被大量拷贝，一旦此节点取消下线，之前已拷贝的大量块会成为多余的副本块。
- ❑ 人为重新设置块副本数。还是以 A 副本块举例，A 副本块当前满足标准副本数 3 个，此时用户张三通过使用 HDFS 中设置副本数 API，人为设置 A 副本数为 1 个，也会造成 A 副本数多于期望值 2 个的情况。
- ❑ 新添加的块记录在系统中被丢失。这种情况相对于前两种的情况，是内部因素造成的。这些新添加的丢失的块记录会在 BlockManager 中再次扫描检测，防止出现多余副本的现象。

以上 3 种情况是可能发生多余副本块的潜在场景。至于这三种情况是如何一步步调用处理多余副本块的过程，下文会一一进行讲述，先来看多余副本块是如何被选出并处理的。

4.2.2 OverReplication 多余副本块处理

多余副本块的处理分为两个子过程：

- ❑ 多余副本块的选出
- ❑ 多余副本块的处理

我们从源码中寻找答案，首先是副本块的选出。

1. 多余副本块的选择

进入 BlockManager 的 processOverReplicatedBlock 方法，此方法名已经表明了其操作的本意。

```
/**
```

```

* 寻找是否有节点包含了多余的副本块，如果确实包含了，则调用
* chooseExcessReplicates 方法标记它们到对象 excessReplicateMap 中。
*/
private void processOverReplicatedBlock(final Block block, final short replication,
    final DatanodeDescriptor addedNode, DatanodeDescriptor delNodeHint) {

```

此方法中注释的意思是找出存在多余副本的节点，如果它们是，则调用 `chooseExcessReplicates` 方法并标记它们，并加入到 `excessReplicateMap` 对象中。下面进行细节的处理：

```

// 节点列表变量的声明
Collection<DatanodeStorageInfo> nonExcess = new ArrayList<DatanodeStorageInfo>();
// 从 corruptReplicas 变量中获取是否存在包含坏块的节点
Collection<DatanodeDescriptor> corruptNodes = corruptReplicas.getNodes(block);

```

继续后面的处理：

```

// 遍历此过量副本块所在的节点列表
for(DatanodeStorageInfo storage : blocksMap.getStorages(block, State.NORMAL)) {
    final DatanodeDescriptor cur = storage.getDatanodeDescriptor();
    ...
    LightweightLinkedSet<Block> excessBlocks = excessReplicateMap.get(cur
        .getDatanodeUuid());
    // 如果在当前多余副本对象 excessReplicateMap 中不存在
    if (excessBlocks == null || !excessBlocks.contains(block)) {
        // 并且所在节点不是已下线或下线中的节点
        if (!cur.isDecommissionInProgress() && !cur.isDecommissioned()) {
            // 并且这个副本块不是损坏的副本块
            // exclude corrupt replicas
            if (corruptNodes == null || !corruptNodes.contains(cur)) {
                // 将此多余副本块的一个所在节点加入候选节点列表中
                nonExcess.add(storage);
            }
        }
    }
}

```

从这里可以看出 `nonExcess` 对象其实是一个候选节点的概念，将块副本所在的节点列表进行多种条件的再判断和剔除。最后再调用选择最终多余副本块节点的方法：

```

chooseExcessReplicates(nonExcess, block, replication,
    addedNode, delNodeHint, blockplacement);

```

进入 `chooseExcessReplicates` 方法：

```

// 首先会形成机架对 DataNode 节点的映射关系图
BlockCollection bc = getBlockCollection(b);
final BlockStoragePolicy storagePolicy = storagePolicySuite.getPolicy(bc,
    getStoragePolicyID());
final List<StorageType> excessTypes = storagePolicy.chooseExcess(
    replication, DatanodeStorageInfo.toStorageTypes(nonExcess));

```

```

// 初始化机架 -> 节点列表映射图对象
final Map<String, List<DatanodeStorageInfo>> rackMap
    = new HashMap<String, List<DatanodeStorageInfo>>();
// 超过一个副本数的节点列表
final List<DatanodeStorageInfo> moreThanOne = new ArrayList<DatanodeStorageInfo>();
// 恰好一个副本数的节点列表
final List<DatanodeStorageInfo> exactlyOne = new ArrayList<DatanodeStorageInfo>();

```

为什么这里要划分出不同的节点列表呢？因为在这里设计者做了优先选择，在同样拥有多余副本块的节点列表中，优先选择节点中副本数多于一个的，其次是副本数恰好为一个的节点。这个设计很好理解，因为上面的多余副本数更多，这里当然要先从多的开始删。

```

// 节点划分成对应两个集合
// moreThanOne 包含了此机架下拥有一个以上副本的节点
// exactlyOne 则包含了余下的节点
replicator.splitNodesWithRack(nonExcess, rackMap, moreThanOne, exactlyOne);

```

进入划分方法：

```

public void splitNodesWithRack(
    final Iterable<DatanodeStorageInfo> storages,
    final Map<String, List<DatanodeStorageInfo>> rackMap,
    final List<DatanodeStorageInfo> moreThanOne,
    final List<DatanodeStorageInfo> exactlyOne) {
    // 遍历候选节点列表，形成机架 -> 节点列表的对应关系
    for(DatanodeStorageInfo s: storages) {
        final String rackName = getRack(s.getDatanodeDescriptor());
        List<DatanodeStorageInfo> storageList = rackMap.get(rackName);
        if (storageList == null) {
            storageList = new ArrayList<DatanodeStorageInfo>();
            rackMap.put(rackName, storageList);
        }
        storageList.add(s);
    }
}

```

下面是划分算法：

```

// 拆分节点到两个集合
for(List<DatanodeStorageInfo> storageList : rackMap.values()) {
    if (storageList.size() == 1) {
        // 如果机架中对应节点的副本存储位置数量只有一个，则表明节点上副本数就为 1，加入 exactlyOne
        // 集合内，否则就为多个，此时加入集合 moreThanOne
        exactlyOne.add(storageList.get(0));
    } else {
        moreThanOne.addAll(storageList);
    }
}
}

```

DatanodeStorageInfo 类表示的是 DataNode 中存放数据的目录位置信息。如果一个 storageList 的大小为 1，说明此节点内只有一个存储此副本的位置信息，间接说明此节点

内只有一个此副本块。于是在这段代码过后，节点组就被分为了两大类：exactlyOne 和 moreThanOne。至此 chooseExcessReplicates 的上半段代码执行完毕，接下来看下半段代码的执行过程：

```
// 选择一个待删除的节点，会偏向 delNodeHintStorage 的节点
// 否则会从节点列表中选出一个可用空间最小的节点
boolean firstOne = true;
final DatanodeStorageInfo delNodeHintStorage
    = DatanodeStorageInfo.getDatanodeStorageInfo(nonExcess, delNodeHint);
final DatanodeStorageInfo addedNodeStorage
    = DatanodeStorageInfo.getDatanodeStorageInfo(nonExcess, addedNode);
```

上面两行注释传达出两个意思：

- 可以直接传入要删除的节点，如果条件允许，则优先选择传入的 delHint 节点。
- 在每个节点的内部列表中，优选选择出可用空间最少的。这个也好理解，在同样拥有过量副本数的节点列表中，选择可用空间尽可能少的，这样可以快速地释放出更多的可用空间。

```
// 如果目前多余副本所在节点数大于标准副本数，则进行循环移除
while (nonExcess.size() - replication > 0) {
    final DatanodeStorageInfo cur;
    // 判断是否可以使用 delNodeHintStorage 节点进行代替
    if (useDelHint(firstOne, delNodeHintStorage, addedNodeStorage,
        moreThanOne, excessTypes)) {
        cur = delNodeHintStorage;
    } else {
        // 否则进行常规的节点选择
        cur = replicator.chooseReplicaToDelete(bc, b, replication,
            moreThanOne, exactlyOne, excessTypes);
    }
}
```

判断是否可以使用 delNodeHintStorage 节点的处理逻辑这里先略过，主要看一下 BlockPlacementPolicy 的 chooseReplicaToDelete 方法，这个分支处理才是最经常用到的处理方式。

```
// 选择的节点要么是心跳时间最老的或者是可用空间最少的
// 从两个集合中按顺序遍历节点
for(DatanodeStorageInfo storage : pickupReplicaSet(first, second)) {
    if (!excessTypes.contains(storage.getStorageType())) {
        continue;
    }
}
```

这里的 first 和 second 分别代表拥有多余一个副本数和恰好拥有一个副本数的节点集合，此处优先选择前者，节点集合选择逻辑如下：

```
protected Collection<DatanodeStorageInfo> pickupReplicaSet(
```

```

    Collection<DatanodeStorageInfo> first,
    Collection<DatanodeStorageInfo> second) {
    return first.isEmpty() ? second : first;
}

```

在节点列表每次的迭代循环中会进行下面两个指标的比较:

```

// 进行心跳时间的对比
if (lastHeartbeat < oldestHeartbeat) {
    oldestHeartbeat = lastHeartbeat;
    oldestHeartbeatStorage = storage;
}
// 进行可用空间的对比
if (minSpace > free) {
    minSpace = free;
    minSpaceStorage = storage;
}

```

然后进行选择, 优先选择心跳时间最老的:

```

final DatanodeStorageInfo storage;
if (oldestHeartbeatStorage != null) {
    storage = oldestHeartbeatStorage;
} else if (minSpaceStorage != null) {
    storage = minSpaceStorage;
} else {
    return null;
}

```

然后进行下面两个操作:

```

// 重新进行 rackMap 对象关系的调整
replicator.adjustSetsWithChosenReplica(rackMap, moreThanOne,
    exactlyOne, cur);
// 将选出的节点从候选节点列表中移除
nonExcess.remove(cur);

```

可以说到了这里, 多余副本块所在节点就被选出了。

2. 多余副本块的处理

此时, 多余副本块的处理就显得比较简洁了, 反正目标对象以及所在节点已经被找到了, 加入到相应的对象中即可:

```

// 加入到 excessReplicateMap 对象中
addToExcessReplicate(cur.getDatanodeDescriptor(), b);

// 将此节点上的多余副本块加入到无效节点中
addToInvalidates(b, cur.getDatanodeDescriptor());

```

加入到 invalidates 无效块列表后不久, 此块就将被清除。

4.2.3 多余副本块清除的场景调用

重新回到上文提到过的多余副本块的三大调用场景。通过查看 `chooseExcessReplicates` 方法的调用就可以找到这些场景，如图 4-3 所示。

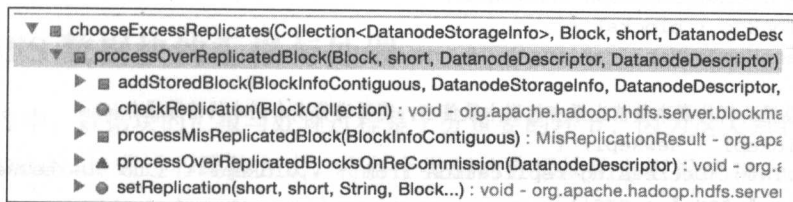


图 4-3 `chooseExcessReplicates` 方法场景调用

针对上述 5 种调用情况，笔者将其归纳为以下 4 类使用场景。

场景 1: ReCommission 重新上线过程

在 `DecommissionManager` 停止下线方法中调用了 `BlockManager` 的 `processOverReplicatedBlocksOnReCommission` 方法来清除多余副本块。

```
// 对指定节点进行下线操作
@VisibleForTesting
public void stopDecommission(DatanodeDescriptor node) {
    if (node.isDecommissionInProgress() || node.isDecommissioned()) {
        // 更新 HeartbeatManager 中的关于此节点的信息
        hbManager.stopDecommission(node);
        // 删除之前下线过程中复制的多余副本
        if (node.isAlive()) {
            blockManager.processOverReplicatedBlocksOnReCommission(node);
        }
        // 在 DecommissionManager 中将此节点移除
        pendingNodes.remove(node);
        decomNodeBlocks.remove(node);
    } else {
        LOG.trace("stopDecommission: Node {} in {}, nothing to do." +
            node, node.getAdminState());
    }
}
```

下线操作重新恢复，会停止正在下线的动作，所以会在这个方法中进行调用。

场景 2: SetReplication 人为设置副本数

人为设置副本数是一个主动因素，调用的方法如下：

```
// 为指定块设置新的副本数
public void setReplication(final short oldRepl, final short newRepl,
    final String src, final Block... blocks) {
```



```

// 如果新设置的副本数与旧的一样，则无须进行任何操作，直接返回
if (newRepl == oldRepl) {
    return;
}

// 更新 needReplication 内部的块队列
for(Block b : blocks) {
    updateNeededReplications(b, 0, newRepl-oldRepl);
}
// 当设置的新的副本数值比原有的副本数值小，需要进行多余副本的清除操作
if (oldRepl > newRepl) {
    LOG.info("Decreasing replication from " + oldRepl + " to " + newRepl
        + " for " + src);
    for(Block b : blocks) {
        processOverReplicatedBlock(b, newRepl, null, null);
    }
} else { // replication factor is increased
    LOG.info("Increasing replication from " + oldRepl + " to " + newRepl
        + " for " + src);
}
}
}

```

这个 API 方法可以被外部的客户端程序调用触发。

场景 3：丢失新添加的块记录信息

当遍历块的时候，有可能会发生丢失新添加块记录的情况。丢失新添加的块信息会导致集群中存在多余的副本。

因为存在丢失块信息的可能性，所以会使用单独的线程重新检测是否存在多余副本的现象：

```

private void processMisReplicatesAsync() throws InterruptedException {
    ...
    while (namesystem.isRunning() && !Thread.currentThread().isInterrupted()) {
        int processed = 0;
        namesystem.writeLockInterruptibly();
        try {
            while (processed < numBlocksPerIteration && blocksItr.hasNext()) {
                BlockInfoContiguous block = blocksItr.next();
                // 此操作中会有检测多余副本块的过程
                MisReplicationResult res = processMisReplicatedBlock(block);
                ...
            }
        } catch (InterruptedException e) {
            // ...
        }
    }
}

```

场景 4：其他场景的检测

其他场景有的时候也会调用 `processOverReplicatedBlock` 方法，但不是外界的因素导致，而是出于一种谨慎性的考虑。比如在 `addStoredBlock` 方法中，当新添加的块被加入到 `blockMap` 中时，会再次进行块的检测。还有一种情况是在文件最终写入完成的时候，也会

调用一次 `checkReplication` 方法，以确认集群中没有多余相同块的情况。这两种情况的调用方法与前面类似，这里就不给出具体的代码了。由此可见，HDFS 的设计者在细节方面的处理真的很用心。

4.3 HDFS 数据块的汇报与处理

在 HDFS 中，数据块的汇报与处理过程是十分重要的环节，因为这关系到整个集群数据的更新。DataNode 通过心跳的方式，将各个类型的数据块信息汇报给 NameNode，然后接收 NameNode 的回复命令。在 NameNode 的处理过程中，这些块会被分为好几种类型，不同类型的块会对应不同的处理逻辑。了解此过程将有助于集群使用者日后的问题排查，通过块操作的输出记录来判断集群当前的运行状况。本节主要介绍对于各个类型块的处理过程，里面有很多细节是非常重要的。这些块包括 5 大类型：新添加的块、待移除的块、无效的块、损坏的块以及正在构建中的块。

4.3.1 块处理的五大类型

如前言中所述，DataNode 的块汇报是在心跳的过程中进行的，也就是下面所示的代码：

```
// 心跳汇报方法
private void offerService() throws Exception {
    ...

    // 心跳汇报的循环执行
    while (shouldRun()) {
        try {
            ...

            // DataNode 执行块汇报动作，并获取 NameNode 返回的回复命令
            List<DatanodeCommand> cmds = blockReport();
            // DataNode 处理回复命令
            processCommand(cmds == null ? null : cmds.toArray(new DatanodeCommand[cmds.
                size()]));
            ...
        }
    }
}
```

那么现在有个问题出现了，DataNode 在做块汇报的时候是每次做全量的汇报还是做增量的汇报？如果先不看后面的代码，我们当然认为做增量的块汇报无疑是更优的选择。后面的代码也验证了我们的这个想法。

```
List<DatanodeCommand> blockReport() throws IOException {
    // 如果时间已经过去，则准备发送新的块报告
    final long startTime = scheduler.monotonicNow();
    if (!scheduler.isBlockReportDue()) {
        return null;
    }
}
```

```

final ArrayList<DatanodeCommand> cmds = new ArrayList<DatanodeCommand>();

// 报告新添加的块以及删除的块
reportReceivedDeletedBlocks();
lastDeletedReport = startTime;
...

```

在每次块汇报的动作中，它将会汇报新接收到的块以及被删除的块，而这两部分的块就是我们所说的增量块的汇报，英文简称 IBR (IncrementBlockReport)。倘若 DataNode 采用每次全量块汇报的方式，NameNode 恐怕也承受不住同时这么多块信息的汇报。在 blockReport 方法的后续调用过程中，最后会调用到 BlockManager 内部的 processReport 块处理方法，如下所示：

```

// 首次块上报处理方法
if (storageInfo.getBlockReportCount() == 0) {
    // 首次块汇报操作，首次块的处理将会比较高效，因为块的类型少
    processFirstBlockReport(storageInfo, newReport);
} else {
    // 正常块处理方法
    invalidatedBlocks = processReport(storageInfo, newReport);
}

```

这里的块处理方法分为两类的原因在于首次块上报的特殊性，因为首次块上报基本都是新增的有效块，在处理效率上比普通块的上报要高很多。我们主要关注正常情况下的块处理过程，也就是 else 逻辑中的 processReport 方法。这个方法的注释很好地概括了这个方法所要做的事情：

```

private Collection<Block> processReport(
    final DatanodeStorageInfo storageInfo,
    final BlockListAsLongs report) throws IOException {
    // 根据旧的与新的块报告之间的不同，修改 block map 中维护的数据关系
    ...
}

```

注释的意思是根据新汇报上来的块报告，适当进行对应关系的修改。假设你之前没有阅读过这部分的代码，可能马上会联想到的上报块的类型只有两种，一种是新添加的块 addedBlock，另一种是需要删除的块：removedBlock 或 deletedBlock。显然 HDFS 在设计的时候不会这么简单，processReport 的头几行代码就表明了到底有多少种类型的块列表。

```

private Collection<Block> processReport(
    final DatanodeStorageInfo storageInfo,
    final BlockListAsLongs report) throws IOException {
    // 根据旧的与新的块报告之间的不同，修改 block map 中维护的数据关系
    // 新添加的块

```

```

Collection<BlockInfoContiguous> toAdd = new LinkedList<BlockInfoContiguous>();
// 待移除的块
Collection<Block> toRemove = new TreeSet<Block>();
// 无效的块
Collection<Block> toInvalidate = new LinkedList<Block>();
// 损坏的块
Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>();
// 正在构建中的块
Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>();
...

```

在这五大类型的块中，第一个和最后一个还是比较好理解的（最后一个的 toUC 是 toUnderConstruction 的缩写），而中间三个则会让人有点混淆。toRemove、toInvalidate 和 toCorrupt 不都表示这些块是待删除的意思吗？这个问题在本章后面的内容中会一一进行分析。图 4-4 为块类型图。

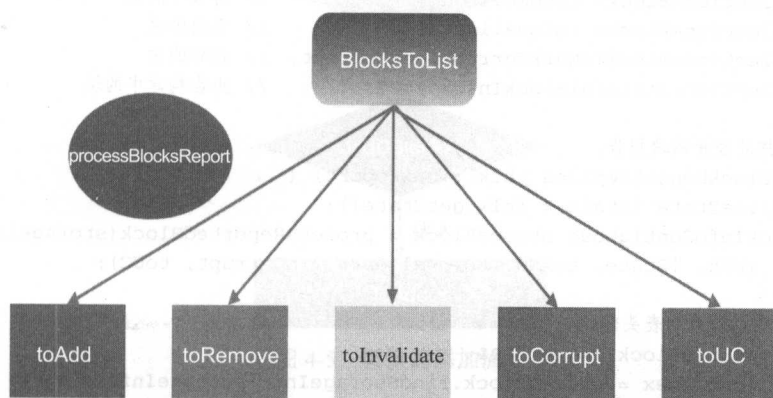


图 4-4 处理块的类型

4.3.2 toAdd : 新添加的块

新添加的块是指那些新复制完的块，而新复制完的块的特征是它的副本状态是 Finalized 的。换句话说，这些块在过去的心跳时间间隔内已完成了块的写操作，并执行完了 finalize 确认动作。但是这里会有一个问题，DataNode 在上报块的时候，是不细分这些块是损坏的或者是正在构建中的，所以就自然地移到了 NameNode 进行这些处理，而比较的方法则是新的块报告与当前维护的块信息之间的对比。现进入 processReport 接下来的内部处理逻辑中：

```

private Collection<Block> processReport(
    final DatanodeStorageInfo storageInfo,
    final BlockListAsLongs report) throws IOException {

```

```

// 根据旧的与新的块报告之间的不同，修改 block map 中维护的数据关系
Collection<BlockInfoContiguous> toAdd = new LinkedList<BlockInfoContiguous>();
Collection<Block> toRemove = new TreeSet<Block>();
Collection<Block> toInvalidate = new LinkedList<Block>();
Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>();
Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>();
// 取出报告中存在差异的部分
reportDiff(storageInfo, report,
           toAdd, toRemove, toInvalidate, toCorrupt, toUC);
...

```

reportDiff 方法，有点像 git diff 命令，“取不同”的意思。在这个方法中，这些列表都以参数的形式传入，reportDiff 方法结束后，这些变量都将被赋值。进入 reportDiff 方法：

```

private void reportDiff(DatanodeStorageInfo storageInfo,
                       BlockListAsLongs newReport,
                       Collection<BlockInfoContiguous> toAdd,      // 新添加的块
                       Collection<Block> toRemove,                 // 待移除的块
                       Collection<Block> toInvalidate,              // 无效的块
                       Collection<BlockToMarkCorrupt> toCorrupt,    // 损坏的块
                       Collection<StatefulBlockInfo> toUC) {        // 正在构建中的块
    ...
    // 扫描处理新的块报告
    for (BlockReportReplica iblk : newReport) {
        ReplicaState iState = iblk.getState();
        BlockInfoContiguous storedBlock = processReportedBlock(storageInfo,
                                                                iblk, iState, toAdd, toInvalidate, toCorrupt, toUC);

        // 移动块到列表头部
        if (storedBlock != null &&
            (curIndex = storedBlock.findStorageInfo(storageInfo)) >= 0) {
            headIndex = storageInfo.moveBlockToHead(storedBlock, curIndex, headIndex);
        }
    }
    ...
}

```

在上面这个 for 循环中，会进行新块的扫描，processReportedBlock 方法中新块的添加过程如图 4-5 所示。

经过重重的逻辑判断，最后会执行添加块的逻辑。如果图 4-5 中的某个判断满足条件，将会提前返回，使之无法成为新的块。比如下面这个判断处理：

```

if (shouldPostponeBlocksFromFuture &&
    namesystem.isGenStampInFuture(block)) {
    queueReportedBlock(storageInfo, block, reportedState,
                      QUEUE_REASON_FUTURE_GENSTAMP);
    return null;
}

```

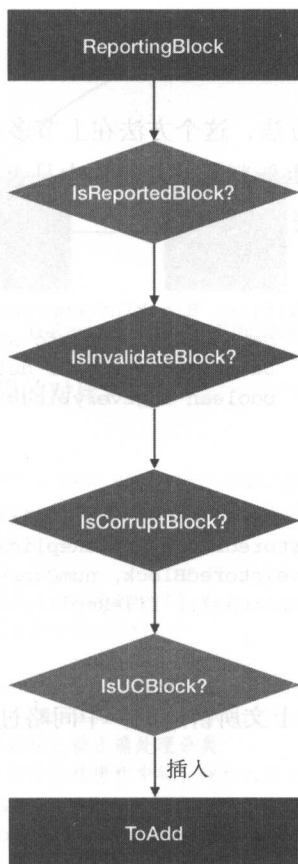


图 4-5 新块的添加流程

经过这个方法处理之后，toAdd 列表中就会多许多的新块，然后返回 reportDiff 所在的 processReport 方法。

```

private Collection<Block> processReport(
    final DatanodeStorageInfo storageInfo,
    final BlockListAsLongs report) throws IOException {
    // 根据旧的与新的块报告之间的不同，修改 block map 中维护的数据关系
    Collection<BlockInfoContiguous> toAdd = new LinkedList<BlockInfoContiguous>();
    Collection<Block> toRemove = new TreeSet<Block>();
    Collection<Block> toInvalidate = new LinkedList<Block>();
    Collection<BlockToMarkCorrupt> toCorrupt = new LinkedList<BlockToMarkCorrupt>();
    Collection<StatefulBlockInfo> toUC = new LinkedList<StatefulBlockInfo>();
    reportDiff(storageInfo, report,
        toAdd, toRemove, toInvalidate, toCorrupt, toUC);
    ...
    for (BlockInfoContiguous b : toAdd) {
        addStoredBlock(b, storageInfo, null, numBlocksLogged < maxNumBlocksToLog);
        numBlocksLogged++;
    }
  }

```

```

    }
    ...
}

```

接着会调用 `addStoredBlock` 方法，这个方法在上节多余副本块的处理内容中已经提到过，它会做一个很关键的动作：重新判断待复制块中是否还存在此块。如果有，则进行移除，及时更新目前的待复制块列表。

```

// 更新内存中维护的块映射信息
private Block addStoredBlock(final BlockInfoContiguous block,
                             DatanodeStorageInfo storageInfo,
                             DatanodeDescriptor delNodeHint,
                             boolean logEveryBlock)
    throws IOException {
    ...
    // 处理多余副本数的情况
    short fileReplication = bc.getBlockReplication();
    if (!isNeededReplication(storedBlock, fileReplication, numCurrentReplica)) {
        neededReplications.remove(storedBlock, numCurrentReplica,
                                   num.decommissionedReplicas(), fileReplication);
    }
    ...
}

```

`toAdd` 块添加的逻辑主要就是上文所讲述的，中间略过了一些操作细节，感兴趣的同学可自行阅读源码进行学习。

4.3.3 toRemove：待移除的块

在日常生活中，我们可能会觉得，移除的意思差不多就是删除的意思，但是在 `processReport` 的块处理方法中，两者并不等同。在移除和删除操作之间还隔了一步操作。在 `NameNode` 这边判断一个块是否为待移除块的标准是它有没有在这一轮被上报上来，如果没有，则表明 `DataNode` 已经把这个块删了。

`toRemove` 的含义是，收集那些没有被汇报上来的块，将这些块放在 `delimiter` 标记块的另一侧。`delimiter` 的意思是分隔符，这个方法的设计思想是让汇报处理过的块和未汇报处理过的块分别位于分隔符的两侧，处理过程如图 4-6 所示。

简单概况这个过程如下：

- 1) 首先会将分隔符块插入到链表头部，这样所有的块默认是未汇报过的。
- 2) 其次遍历块，将汇报过的块移到链表头部，这样块的位置就从分隔符的右侧挪到了左侧。
- 3) 于是在本轮没有被汇报处理过的块就全都在分隔符块的右侧了。

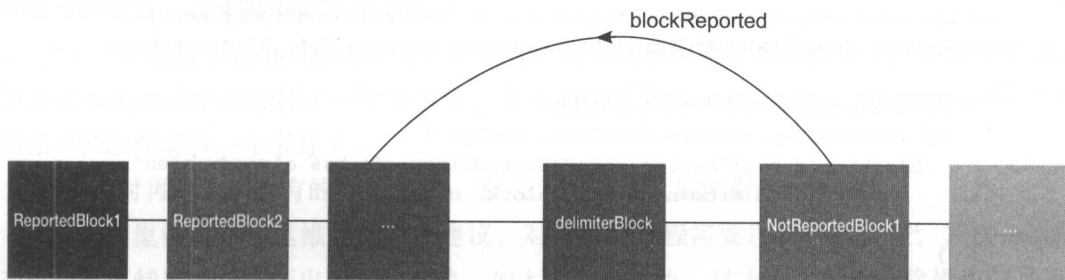


图 4-6 ToRemove 待移除块的处理逻辑图

代码如下，大家可以进行过程的对比：

```
// 新建一个标记块用来区分块
BlockInfoContiguous delimiter = new BlockInfoContiguous(new Block(), (short) 1);
AddBlockResult result = storageInfo.addBlock(delimiter);
...
// 扫描处理新的块报告
for (BlockReportReplica iblk : newReport) {
    ReplicaState iState = iblk.getState();
    // 处理新汇报上来的块
    BlockInfoContiguous storedBlock = processReportedBlock(storageInfo,
        iblk, iState, toAdd, toInvalidate, toCorrupt, toUC);
    // 如果处理结果不为空，则表明此块已被正确处理分类
    // 将此块移到标记块的头部，否则在后续操作中将会被加入到 toRemove 列表中
    if (storedBlock != null &&
        (curIndex = storedBlock.findStorageInfo(storageInfo)) >= 0) {
        headIndex = storageInfo.moveBlockToHead(storedBlock, curIndex, headIndex);
    }
}

// 收集那些没有被报告上来的块
Iterator<BlockInfoContiguous> it =
    storageInfo.new BlockIterator(delimiter.getNext(0));
while(it.hasNext())
    toRemove.add(it.next());
storageInfo.removeBlock(delimiter);
```

一般的块会在 `processReportedBlock` 方法中被处理掉，而那些没有被处理掉的块最后会被加入到 `toRemove` 列表中，`toRemove` 对象于是就被赋值了。然后重新跳回到 `processReport` 方法，观察 `toRemove` 是被如何处理的。

```
for (Block b : toRemove) {
    removeStoredBlock(b, node);
}
```

进入 `removeStoredBlock` 方法：

```
// 更新内存中维护的块映射信息，与 addStoredBlock 类似
```

```

public void removeStoredBlock(Block block, DatanodeDescriptor node) {
    blockLog.debug("BLOCK* removeStoredBlock: {} from {}", block, node);
    ...
    // 从 blocksMap 对象中进行此块的移除
    if (!blocksMap.removeNode(block, node)) {
        blockLog.debug("BLOCK* removeStoredBlock: {} has already been" +
            " removed from node {}", block, node);
        return;
    }
    ...
}

```

这个方法做的一步很重要的操作是将 toRemove 中的块从 blocksMap 映射关系中移除掉。到了这里，终于知道 remove 的具体含义了，remove 指的是 blocksMap 的移除动作。从 blocksMap 中移除掉块之后，会触发其他的行为操作，而并不是一个简简单单的移除动作。

4.3.4 toInvalidate : 无效的块

在 HDFS 中，无效的块在某种程度上来说更接近于待删除块的意思。在 process-ReportedBlock 的方法中，无效块最根本的来源是 blocksMap 中不存在的块。

```

// 根据块 Id 找到对应的块对象
BlockInfoContiguous storedBlock = blocksMap.getStoredBlock(block);
if (storedBlock == null) {
    // 如果 blockMap 对象中无此块对象，则将此块移入无效块列表中，
    // 这个块将从 DataNode 中删除
    toInvalidate.add(new Block(block));
    return null;
}

```

所有合法块的信息都会被记录在 blocksMap 中，所以 blocksMap 对象中不存在块信息的场景无外乎两种：

- ❑ 第一种是刚刚 toRemove 中的块信息，使得 blocksMap 移除了对应的块信息。
- ❑ 第二种是新汇报上来的块信息，DataNode 自身有这些块信息，而 NameNode 自身的 blocksMap 中没有，也会被认为是无效块。

第一种场景好理解，原本就是准备删除的块，就应该移到无效块中。这里要特别注意第二种场景，这个场景笔者在工作中经历过一次，这个场景挺奇怪的，什么情况下 DataNode 有块信息，而 NameNode 中又没有呢？答案是 NameNode 用早些时间的镜像文件做启动操作。因为时间早，镜像文件当然不会有后面的元数据信息，所以这个时候如果启动了 DataNode，简直就是灾难。你会看到大量的块被移到无效块列表中，那什么时候会无意中用了早期的镜像文件启动 NameNode 呢？比如下面这个条件：

- 1) NameNode 是 HA 模式的。
- 2) 某台 namenode1 进程一个月前挂了，此时另外一台 namenode2 切换到了 active 状态

继续提供服务，而集群维护者并不知道。

3) 一个月后 namenode2 需要更改配置或因为别的原因需要重启集群，且集群维护者并不知道 namenode1 已经挂了一个多月了，意外地先启动了 namenode1，而 namenode1 上的镜像文件已经落后一个多月了。

4) 此时再次重启所有的 DataNode，就会发生上述大量块删除的场景。

所以这里给集群的运维人员一个建议，对于关键进程需要进行报警监控，其次每次启动集群时要特别留意是否出现异常现象，如大量的无效块、日志中大量的删除操作记录等。在任何情况下，数据的安全性永远是最重要的。还好上面这种情况的 bug 在较新的版本中已经被解决了，老的镜像文件如果落后太多将会启动失败。图 4-7 是 toInvalidate 无效块的处理流程图。

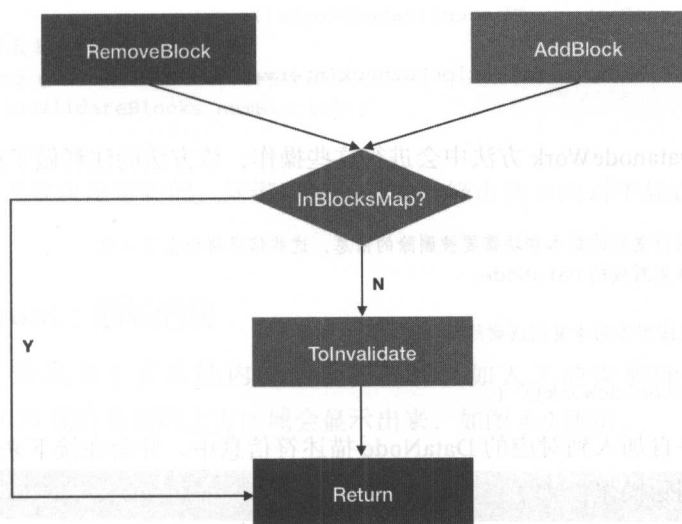


图 4-7 ToInvalidate 无效块处理逻辑图

回到原来的话题，加入到 toInvalidate 块后，toInvalidate 中的块会被加入到 invalidate-Blocks 中：

```

for (Block b : toInvalidate) {
    addToInvalidates(b, node);
}

```

// 将块加入到无效块列表中

```

void addToInvalidates(final Block block, final DatanodeInfo datanode) {
    if (!namesystem.isPopulatingReplQueues()) {
        return;
    }
}

```

```

        invalidateBlocks.add(block, datanode, true);
    }

```

之后就会触发删除操作了。这部分的操作将会在 BlockManager 内部的 Replication-Monitor 监控线程中被调用：

```

private class ReplicationMonitor implements Runnable {
    @Override
    public void run() {
        while (namesystem.isRunning()) {
            try {
                // 只有当 NameNode 退出安全模式后，才能进行接下来的操作
                if (namesystem.isPopulatingReplQueues()) {
                    computeDatanodeWork();
                    processPendingReplications();
                    rescanPostponedMisreplicatedBlocks();
                }
                Thread.sleep(replicationRecheckInterval);
            }
            ...
        }
    }
}

```

在 computeDatanodeWork 方法中会进行这些操作，该方法的注释做了很好的解释：

```

/**
 * 计算块剩余的待复制的副本和块需要被删除的信息，这些信息将会在下一次
 * 的心跳中通知到对应的 DataNode。
 *
 * @return 返回因为副本复制或删除而调度的块的个数
 */
int computeDatanodeWork() {

```

这些无效块各自加入到对应的 DataNode 描述符信息中，并会在接下来的操作中通过心跳通知节点执行删除操作：

```

    int blockCnt = 0;
    for (DatanodeInfo dnInfo : nodes) {
        int blocks = invalidateWorkForOneNode(dnInfo);
        if (blocks > 0) {
            blockCnt += blocks;
            if (--nodesToProcess == 0) {
                break;
            }
        }
    }
}

```

NameNode 50070 端口页面中的 PendingDeletionBlock 计数其实就是 invalidateBlocks 的大小，如图 4-8 所示。

Decommissioning Nodes	0
Total Datanode Volume Failures	1 (0 B)
Number of Under-Replicated Blocks	1
Number of Blocks Pending Deletion	21
Block Deletion Start Time	2016年1月13日 GMT+8下午5:19:30

图 4-8 NameNode 页面的 PendingDeletionBlock 块

相关代码如下：

```
@Override
@Metric
public long getPendingDeletionBlocks() {
    return blockManager.getPendingDeletionBlocksCount();
}
// 待删除块数其实就是无效块的数量
public long getPendingDeletionBlocksCount() {
    return invalidateBlocks.numBlocks();
}
```

toInvalidate 无效块是重要的，知道这些无效块的缘由和去向对于处理问题是非常有帮助的。

4.3.5 toCorrupt : 损坏的块

损坏的块一般发生于非系统内部的损坏操作，如人工的误删除操作。损坏块在 NameNode 的 50070 端口页面的上方区域会显示出来，如图 4-9 所示。

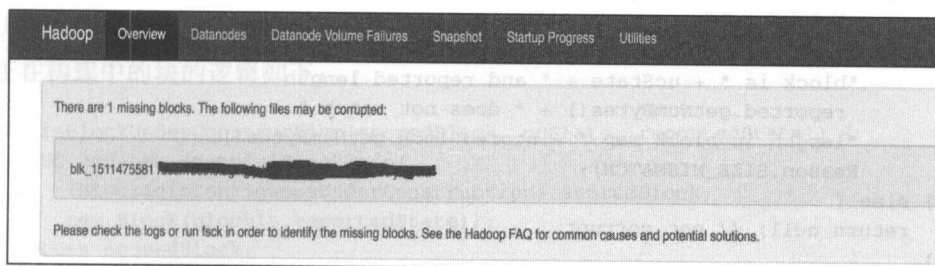


图 4-9 NameNode 页面的损坏块

在 processReportedBlock 的处理逻辑中，对损坏块的判断逻辑主要在 checkReplicaCorrupt 方法中：

```
...
BlockToMarkCorrupt c = checkReplicaCorrupt(
    block, reportedState, storedBlock, ucState, dn);
if (c != null) {
```

```

if (shouldPostponeBlocksFromFuture) {
    // 如果当前块的版本号已经超出了其应有的范围
    // 先加入到报告队列，等后面的时间再处理
    queueReportedBlock(storageInfo, storedBlock, reportedState,
        QUEUE_REASON_CORRUPT_STATE);
} else {
    // 将块加入到损坏块列表中
    toCorrupt.add(c);
}
return storedBlock;
}
...

```

进入 `checkReplicaCorrupt` 方法，你会看到判断一个块是否为损坏状态的两大主要因素：

```

private BlockToMarkCorrupt checkReplicaCorrupt(
    Block reported, ReplicaState reportedState,
    BlockInfoContiguous storedBlock, BlockUCState ucState,
    DatanodeDescriptor dn) {
    switch(reportedState) {
        case FINALIZED:
            switch(ucState) {
                case COMPLETE:
                    case COMMITTED:
                        // 如果块的版本号不同，意味着块的版本信息发生变化
                        if (storedBlock.getGenerationStamp() != reported.getGenerationStamp()) {
                            final long reportedGS = reported.getGenerationStamp();
                            return new BlockToMarkCorrupt(storedBlock, reportedGS,
                                "block is " + ucState + " and reported genstamp " + reportedGS
                                + " does not match genstamp in block map "
                                + storedBlock.getGenerationStamp(), Reason.GENSTAMP_MISMATCH);
                        } else if (storedBlock.getNumBytes() != reported.getNumBytes()) {
                            // 块大小发生变化
                            return new BlockToMarkCorrupt(storedBlock,
                                "block is " + ucState + " and reported length " +
                                reported.getNumBytes() + " does not match " +
                                "length in block map " + storedBlock.getNumBytes(),
                                Reason.SIZE_MISMATCH);
                        } else {
                            return null; // not corrupt
                        }
                    }
            }
        ...
    }
}

```

一个是长度大小不匹配，另一个是版本信息不匹配，这如何理解呢？在 HDFS 中，块在创建完毕之后，会产生一个叫 `generationStamp` 的信息，以后每次块的内容改动，这个值都会向前追加，表示块版本的变更。判断完是否为损坏的块之后，在 `processReport` 的 `markBlockAsCorrupt` 方法中，会将块加入到 `corruptReplicas` 对象里，代码如下：

```

private void markBlockAsCorrupt(BlockToMarkCorrupt b,

```

```

DatanodeStorageInfo storageInfo,
DatanodeDescriptor node) throws IOException {

...

// 将此副本加入到损坏块 map 中
corruptReplicas.addToCorruptReplicasMap(b.corrupted, node, b.reason,
    b.reasonCode);
...

```

`corruptReplicas` 对象中的某些方法信息会在 `fsck` 命令中被用到。仔细留意过 `NameNode` 页面损坏块信息的读者肯定会发现，这些损坏块的信息每次打开时都在，并不会消失。的确是这样的，除非你执行了 `fsck` 的 `-delete` 或 `-move` 操作，将损坏的块彻底删除掉了，从而使元信息也被删除掉。在 `NameNode` 页面上所表示的损坏块的个数就是 `corruptReplicas` 对象的大小。

```

/** Returns number of blocks with corrupt replicas */
@Metric({"CorruptBlocks", "Number of blocks with corrupt replicas"})
public long getCorruptReplicaBlocks() {
    return blockManager.getCorruptReplicaBlocksCount();
}

void updateState() {
    ...
    corruptReplicaBlocksCount = corruptReplicas.size();
}

```

4.3.6 toUC：正在构建中的块

UC 的全称是 `UnderConstruction`，正在构建的意思，表明此块正在进行写动作。判断是否是正在构建中的块的逻辑如下：

```

if (isBlockUnderConstruction(storedBlock, ucState, reportedState)) {
    toUC.add(new StatefulBlockInfo(
        (BlockInfoContiguousUnderConstruction) storedBlock,
        new Block(block), reportedState));
    return storedBlock;
}

// 判断块是否处于正在构建状态的方法
private boolean isBlockUnderConstruction(BlockInfoContiguous storedBlock,
    BlockUCState ucState, ReplicaState reportedState) {
    switch(reportedState) {
    case FINALIZED:
        switch(ucState) {
        case UNDER_CONSTRUCTION:
        case UNDER_RECOVERY:

```

```

        return true;
    default:
        return false;
    }
    case RBW:
    case RWR:
        return (!storedBlock.isComplete());
    // 下面所属副本状态的块不会被汇报上来，不做处理
    case RUR:
    case TEMPORARY:
    default:
        return false;
    }
}

```

块的状态根据传入的副本状态进行判断。所有 UC 的块添加完毕之后，进入 `addStoreBlockUnderConstruction`：

```

// 处理正在构建中的块
for (StatefulBlockInfo b : toUC) {
    addStoredBlockUnderConstruction(b, storageInfo);
}
void addStoredBlockUnderConstruction(StatefulBlockInfo ucBlock,
    DatanodeStorageInfo storageInfo) throws IOException {
    BlockInfoContiguousUnderConstruction block = ucBlock.storedBlock;
    block.addReplicaIfNotPresent(
        storageInfo, ucBlock.reportedBlock, ucBlock.reportedState);
    ...
}

```

在 `addReplicaIfNotPresent` 方法内，这些正在构建中的块信息会加入到 `replicas` 变量列表中：

```

void addReplicaIfNotPresent(DatanodeStorageInfo storage,
    Block block,
    ReplicaState rState) {
    Iterator<ReplicaUnderConstruction> it = replicas.iterator();
    ...
    replicas.add(new ReplicaUnderConstruction(block, storage, rState));
}

```

`replicas` 的定义如下：

```

// 正在构建中的块
private List<ReplicaUnderConstruction> replicas;

```

`toUC` 的相关块处理逻辑相对比较简单，主要过程就是上面所讲述的。

以上就是本节所讲述的上报块处理的五大分支过程，每个分支处理都尽量地把与其相关的操作也纳入了进来。笔者在解读此方面的代码时，略过了一些细节。如果读者想要了

解更加详细的过程，请阅读 BlockManager 中的 processReport 以及其内部 reportDiff 方法。

4.4 小结

本章各个小节总体不会太难，对于 HDFS 的 fsck 命令，我们只要了解如何使用即可。本章的重点和难点主要在块的汇报处理过程，理解此过程将会对我们排除问题非常有帮助。

HDFS 的流量处理

本章将介绍 HDFS 中比较特殊的处理过程：流量处理。Hadoop 集群随着规模的扩增，有的时候每日的吞吐量会很大，此时了解它内部的流量处理细节非常有帮助。本章主要从 HDFS 的内部限流、HDFS 的 Balancer 数据平衡以及 DiskBalancer 磁盘间数据平衡三方面的内容来进行学习。前两部分内容是应用场景中经常碰到的，而 DiskBalancer 则是拓展的内容。

5.1 HDFS 的内部限流

HDFS 内部限流的意思是指在 HDFS 中对数据流量进行限制。在 HDFS 内部，每天的数据吞吐量是巨大的，一个大任务往往会读写超过 GB 级别的数据量。除了运行任务之外，在 HDFS 内部还有其他一些场合会发生数据的传输，如何对这些场合进行流量的限制就显得格外重要了。一旦有大规模数据流量持续地在传输，就有可能影响到服务的正常运行。在本节中，我们将介绍 HDFS 内部的多个限流场景、数据流量的限流原理以及对现有限流方式的一些改进想法。

5.1.1 数据的限流

数据的限流更让人理解的称呼应该是“数据流的限流”。数据流指的是传输中的源源不断的数据。这些数据传输会耗尽大量的网络带宽。一台机器的网络带宽必定是有限的，如果带宽被这台机器上的某些任务占满的话，就会影响正常任务的数据传输。如果带宽长时

间地被占满，还会造成机器网络 IO 报警，所以限流的目的正在于此。可能造成网络带宽迅速被占满的不一定都是恶意的程序或服务，程序中一个疏忽的处理或小错误都可能造成大规模数据的传输，所以与其去劝导用户规范写程序，还不如从系统层面进行管理限制，把主动权掌握在自己手中。

数据限流的涉及面很大，因为数据类型和使用场景就有很多。所以本节只分析我们想要分析的数据限流：Hadoop 内部的限流机制。作为一个大型的分布式存储系统，数据的读写操作往往是非常频繁的，所以数据的传输量一定很大。在数据量传输很大的情况下，如何避免出现个别服务把带宽占满的情况就显得格外重要。有一点是至少需要保证的，即在 Hadoop 中正在运行的任务的读写数据操作都是正常的。为了方便下文的描述，我们暂且称此类型的数据为“普通任务数据流”，当然还存在另外的数据流传输，而且类型比想象中更多：

- Balancer 数据平衡数据流传输。
- Fsimage 镜像文件的上传下载数据流传输。
- VolumeScanner 磁盘扫描时的数据流传输。

看完这 3 个结果，第一个 Balancer 的数据流传输还是能想得到的，后面两个如果没有从源码中进行分析，很容易会忽略掉。因为以上 3 种属于非正常业务的数据流传输，是在系统自身内部进行的，所以 Hadoop 对这 3 种操作做了限流操作。限流相关的类名叫作 `DataTransferThrottler`，图 5-1 为 HDFS 内部的限流结构。

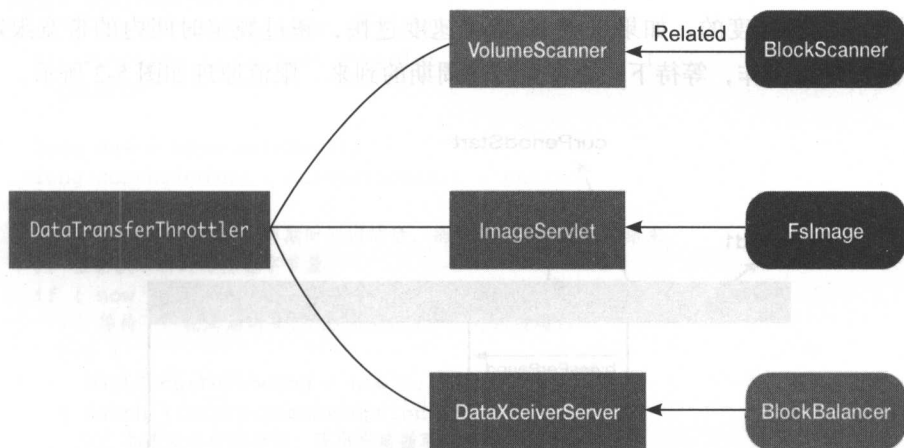


图 5-1 HDFS 内部的限流结构

5.1.2 DataTransferThrottler 限流原理

数据传输的限流原理在 DataTransferThrottler 中有着非常巧妙的设计。先看这个类的源码注释：

```
/**
 * 一个限制数据传输的类。
 * 这个类是线程安全的，它可以被多个线程共享，带宽参数 bandwidthPerSec
 * 指定了所有线程所共享的总带宽大小。
 */
public class DataTransferThrottler
```

通过传入指定的带宽速率来作为一个最大的限制值，在限制类的作用下，带宽的平均速度将会控制在这个速率之下。在这个类中，定义了下面几个变量：

```
// 单位周期时间大小
private final long period;
// 最大可累积的周期时间
private final long periodExtension;
// 每个周期内可允许传输的总字节大小
private long bytesPerPeriod;
// 当前周期的起始时间
private long curPeriodStart;
// 当前周期内剩余可传输字节大小
private long curReserve;
private long bytesAlreadyUsed;
```

在 DataTransferThrottler 类中的主要限流思想是通过单位时间段内限制指定字节数的方式来控制平均传输速度的。如果发现 IO 传输速度过快，超过规定时间内的带宽限定字节数，则会进行等待操作，等待下一个带宽传输周期的到来。限流原理如图 5-2 所示。

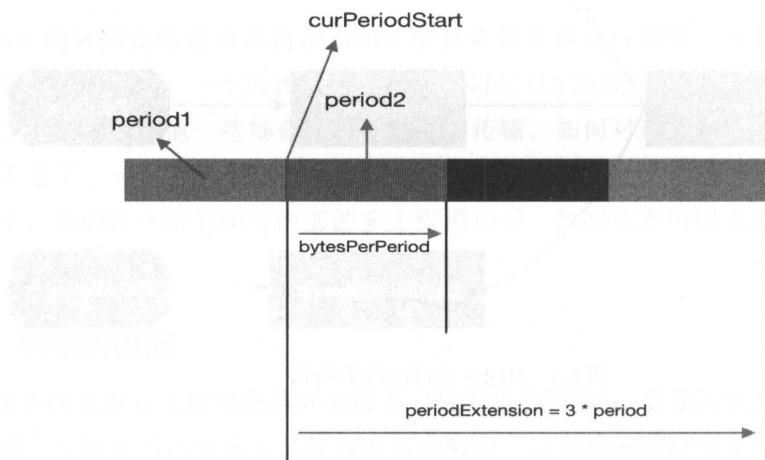


图 5-2 DataTransferThrottler 限流原理

因此每个周期内的可允许传输字节数就是很关键的变量，它是根据传入的带宽上限值进行转换的。

```
// 数据传输限流器对象构造方法
public DataTransferThrottler(long period, long bandwidthPerSec) {
    this.curPeriodStart = monotonicNow();
    this.period = period;
    // 将带宽按照周期做比例转化
    this.curReserve = this.bytesPerPeriod = bandwidthPerSec*period/1000;
    this.periodExtension = period*3;
}
```

因为传入的带宽是以秒为单位的，而周期单位是毫秒，所以要除以 1000。curReserve 这个变量的意思可理解为当前可使用的字节传输量。初始传输值就是一个周期的可传输字节数。DataTransferThrottler 的 throttle 方法是带宽限制的核心方法。

```
public synchronized void throttle(long numOfBytes, Canceler canceler) {
    if ( numOfBytes <= 0 ) {
        return;
    }
}
```

```
// 当前可传输的字节数减去当前发送、接收字节数
```

```
curReserve -= numOfBytes;
```

```
// 当前字节使用量进行相应增加
```

```
bytesAlreadyUsed += numOfBytes;
```

```
// 如果 curReserve 小于等于 0，说明当前周期内可使用字节数已经用完
```

```
while (curReserve <= 0) {
```

```
    // 如果设置了 canceler 对象，则不会进行限流操作
```

```
    if (canceler != null && canceler.isCancelled()) {
```

```
        return;
```

```
    }
```

```
    long now = monotonicNow();
```

```
    long curPeriodEnd = curPeriodStart + period;
```

```
// 如果当前时间还在本周期时间内的话，则必须等待此周期的结束，
```

```
// 重新获取新的可传输字节量
```

```
if ( now < curPeriodEnd ) {
```

```
    // 等待下一轮周期到来，curReserve 才可以被增加
```

```
    try {
```

```
        wait( curPeriodEnd - now );
```

```
    } catch (InterruptedException e) {
```

```
        // 如果发生中断异常，跳出当前循环
```

```
        Thread.currentThread().interrupt();
```

```
        break;
```

```
    }
```

```
} else if ( now < (curPeriodStart + periodExtension)) {
```

```
    // 如果当前时间已经超过此周期的时间且不大于最大周期间隔，则增加可接受字节数
```

```
    // 并更新周期起始时间为前一周期的末尾时间
```

```

        curPeriodStart = curPeriodEnd;
        curReserve += bytesPerPeriod;
    } else {
        // 如果当前时间超过 curPeriodStart + periodExtension, 则表示
        // 已经长时间没有使用 Throttler, 重置时间
        curPeriodStart = now;
        curReserve = bytesPerPeriod - bytesAlreadyUsed;
    }
}

// 传输结束, 当前字节使用量进行移除
bytesAlreadyUsed -= numOfBytes;
}

```

从这里可以得到一个启发, 影响带宽平均传输速率的指标不仅仅只有传入的带宽速度上限值参数, 周期的设置同样也很重要。带宽周期设小了, 发生等待的次数会相对变多, 最后的带宽平均速度就会变低。这个问题在下文中还会继续提到。

5.1.3 数据流限流在 Hadoop 中的使用

了解完 `DataTransferThrottler` 中的限流原理之后, 我们有必要了解 Hadoop 在哪些地方对数据做了限流动作。

1. Balancer

数据 Balancer 平衡的操作, 其中 `Throttler` 限流器对象是在 `DataXceiverServer` 类中创建的。

```

// 初始化 Balancer 限流器
this.balanceThrottler = new BlockBalanceThrottler(
    conf.getLong(DFSConfigKeys.DFS_DATANODE_BALANCE_BANDWIDTHPERSEC_KEY,
        DFSConfigKeys.DFS_DATANODE_BALANCE_BANDWIDTHPERSEC_DEFAULT),
    conf.getInt(DFSConfigKeys.DFS_DATANODE_BALANCE_MAX_NUM_CONCURRENT_MOVES_KEY,
        DFSConfigKeys.DFS_DATANODE_BALANCE_MAX_NUM_CONCURRENT_MOVES_DEFAULT));

```

下面这个 Balancer 带宽大小配置属性就是设置给 `Throttler` 对象的。

```

public static final String DFS_DATANODE_BALANCE_BANDWIDTHPERSEC_KEY = "dfs.
    datanode.balance.bandwidthPerSec";
public static final long DFS_DATANODE_BALANCE_BANDWIDTHPERSEC_DEFAULT =
    1024*1024;

```

默认带宽大小 1MB。这个 `Throttler` 对象在 `DataXceiver` 的 `replaceBlock` 和 `copyBlock` 方法中被调用。

```

@Override
public void copyBlock(final ExtendedBlock block,
    final Token<BlockTokenIdentifier> blockToken) throws IOException {

```

```

...

long beginRead = Time.monotonicNow();
// 在读取块数据时, 传入 Balancer 限流器对象进行限流
long read = blockSender.sendBlock(reply, baseStream,
                                   dataXceiverServer.balanceThrottler);

long duration = Time.monotonicNow() - beginRead;
datanode.metrics.incrBytesRead((int) read);
datanode.metrics.incrBlocksRead();
datanode.metrics.incrTotalReadTime(duration);

...

@Override
public void replaceBlock(final ExtendedBlock block,
                        final StorageType storageType,
                        final Token<BlockTokenIdentifier> blockToken,
                        final String delHint,
                        final DatanodeInfo proxySource) throws IOException {
    ...

    // 在接收块时, 传入 Balancer 限流器对象进行限流
    blockReceiver.receiveBlock(null, null, replyOut, null,
                               dataXceiverServer.balanceThrottler, null, true);

    // 通知 NameNode 已接收到块
    datanode.notifyNamenodeReceivedBlock(
        block, delHint, blockReceiver.getStorageUuid());

    LOG.info("Moved " + block + " from " + peer.getRemoteAddressString()
            + ", delHint=" + delHint);
}
...

```

最后会调用 BlockSender 的 sendPacket 和 BlockReceiver 的 receivePacket 方法, 分别在 BlockSender、BlockReceiver 类的下面两个方法中调用 throttle 的方法:

```

private int sendPacket(ByteBuffer pkt, int maxChunks, OutputStream out,
                      boolean transferTo, DataTransferThrottler throttler) throws IOException {
    int dataLen = (int) Math.min(endOffset - offset,
                                (chunkSize * (long) maxChunks));
    // 将一个数据包拆分为多个 chunk 单元发送, 下面是计算需要发送的 chunk 数量
    int numChunks = numberOfChunks(dataLen);
    int checksumDataLen = numChunks * checksumSize;
    int packetLen = dataLen + checksumDataLen + 4;
    boolean lastDataPacket = offset + dataLen == endOffset && dataLen > 0;

    ...

    // 如果限流器对象不为空, 则表明需要进行限流操作
    if (throttler != null) {
        throttler.throttle(packetLen);
    }
}

```

```

    return dataLen;
}

private int receivePacket() throws IOException {
    // 读取下一个数据包
    packetReceiver.receiveNextPacket(in);

    ...

    // 限流原理同上
    if (throttler != null) {
        throttler.throttle(len);
    }

    return lastPacketInBlock?-1:len;
}

```

因此可以从侧面了解到 DataXceiver 的 copyBlock 和 replaceBlock 方法都是在处理 Balancer 数据平衡相关程序时使用的。

2. TransferFsImage

TransferFsImage 指的是镜像文件的上传下载过程。可能是 Hadoop 的设计者考虑到经常性的镜像文件的传输对集群短时间内的带宽也会有所影响，因此也进行了带宽限制的操作。上传与下载镜像文件的过程比较类似，以下载镜像文件为例子：

```

@Override
protected void doPut(final HttpServletRequest request,
    final HttpServletResponse response) throws ServletException, IOException {
    try {
        ServletContext context = getServletContext();
        final FSImage nnImage = NameNodeHttpServer.getFsImageFromContext(context);
        final Configuration conf = (Configuration) getServletContext()
            .getAttribute(JspHelper.CURRENT_CONF);
        final PutImageParams parsedParams = new PutImageParams(request, response,
            conf);
        final NameNodeMetrics metrics = NameNode.getNameNodeMetrics();

        validateRequest(context, conf, request, response, nnImage,
            parsedParams.getStorageInfoString());

        UserGroupInformation.getCurrentUser().doAs(
            new PrivilegedExceptionAction<Void>() {

                @Override
                public Void run() throws Exception {

                    ...

                    InputStream stream = request.getInputStream();

```

```

try {
    long start = monotonicNow();
    // 此处传入限流器对象
    MD5Hash downloadImageDigest = TransferFsImage
        .handleUploadImageRequest(request, txid,
            nnImage.getStorage(), stream,
            parsedParams.getFileSize(), getThrottler(conf));
    ...
}

```

其中 `getThrottler` 方法会从配置文件的相关属性中得到此限流器实例对象：

```

// 得到镜像文件传输限流器对象
public final static DataTransferThrottler getThrottler(Configuration conf) {
    long transferBandwidth =
        conf.getLong(DFSConfigKeys.DFS_IMAGE_TRANSFER_RATE_KEY,
            DFSConfigKeys.DFS_IMAGE_TRANSFER_RATE_DEFAULT);
    DataTransferThrottler throttler = null;
    if (transferBandwidth > 0) {
        throttler = new DataTransferThrottler(transferBandwidth);
    }
    return throttler;
}

```

默认返回的 `throttler` 限流器对象为 `null`，因为限制带宽默认为 0：

```

public static final String DFS_IMAGE_TRANSFER_RATE_KEY =
    // 此默认值表明默认不限流
    "dfs.image.transfer.bandwidthPerSec";
public static final long DFS_IMAGE_TRANSFER_RATE_DEFAULT = 0;

```

最终在 `receiveFile` 方法中调用了 `throttle` 方法：

```

private static MD5Hash receiveFile(String url, List<File> localPaths,
    Storage dstStorage, boolean getChecksum, long advertisedSize,
    MD5Hash advertisedDigest, String fsImageName, InputStream stream,
    DataTransferThrottler throttler) throws IOException {
    ...

    int num = 1;
    byte[] buf = new byte[HdfsConstants.IO_FILE_BUFFER_SIZE];
    while (num > 0) {
        num = stream.read(buf);
        if (num > 0) {
            received += num;
            for (FileOutputStream fos : outputStreams) {
                fos.write(buf, 0, num);
            }
            // 如果限流器对象不为空，则表明此处需要限流
            if (throttler != null) {
                throttler.throttle(num);
            }
        }
    }
}

```

此处限流器是默认不开启的。

3. VolumeScanner

VolumeScanner 的意思是磁盘扫描，磁盘扫描的目的是为了发现坏的块。坏的块一般发生在读操作异常的情况下，所以这个阶段读的块会被列为可疑块。Hadoop 设计者为了确保本节点的正常 IO 不受影响，特意对磁盘扫描的带宽做了预先限制，防止这样一个附属操作服务影响到正常业务。限流操作在 VolumeScanner 的 scanBlock 方法中被调用：

```
// 扫描块操作
private long scanBlock(ExtendedBlock cblock, long bytesPerSec) {
    ...
    BlockSender blockSender = null;
    try {
        blockSender = new BlockSender(block, 0, -1,
            false, true, true, datanode, null,
            CachingStrategy.newDropBehind());
        // 为限流器对象设置带宽速率
        throttler.setBandwidth(bytesPerSec);
        // 传入限流器对象进行限流
        long bytesRead = blockSender.sendBlock(nullStream, null, throttler);
        resultHandler.handle(block, null);
        return bytesRead;
    }
    ...
}
```

bytesPerSec 带宽速率值在下面这个方法中被赋值：

```
@SuppressWarnings("unchecked")
Conf(Configuration conf) {
    this.targetBytesPerSec = Math.max(0L, conf.getLong(
        DFS_BLOCK_SCANNER_VOLUME_BYTES_PER_SECOND,
        DFS_BLOCK_SCANNER_VOLUME_BYTES_PER_SECOND_DEFAULT));
    ...
}

public static final String DFS_BLOCK_SCANNER_VOLUME_BYTES_PER_SECOND = "dfs.
    block.scanner.volume.bytes.per.second";
public static final long DFS_BLOCK_SCANNER_VOLUME_BYTES_PER_SECOND_DEFAULT = 1048576L;
```

上面代码的限流大小默认是 1MB，限流部分的操作就是以上 3 个部分。图 5-3 为 HDFS 内部限流执行图。

5.1.4 Hadoop 限流优化点

学习完整个限流部分的代码之后，可以看到很多设计的巧妙之处，但是同样存在美中不足的地方，笔者总结出了其中两点：

1) DataTransferThrottler 的周期时间存在硬编码的现象，周期长短的设置对于带宽的影响也不容忽视，原因在上文已经提到过。目前的设置是 500 毫秒。

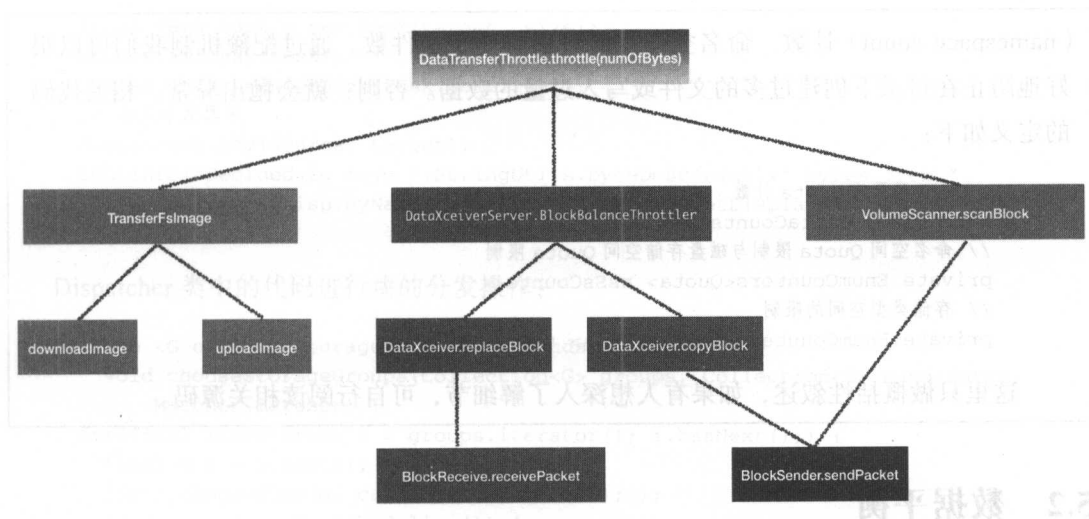


图 5-3 HDFS 内部限流详细执行图

```

public DataTransferThrottler(long bandwidthPerSec) {
    // 默认限流周期 500 毫秒
    this(500, bandwidthPerSec);
}

```

优化建议是将其配置化处理，对于这点笔者在学习的过程中已建立相关 Issue，并提交给了开源社区，编号为 HDFS-9756。Patch 代码链接如下：

<https://github.com/linyiqun/open-source-patch/tree/master/hdfs/HDFS-9756>

2) 上述带宽限制的场景都有一个共同点，即都还只是在非任务层面做限制，并没有在正常的读写块操作上做限制。这样的话，任务的数据传输将会耗尽已有带宽。笔者认为可以把这方面的限制也加上，做成可配置的，默认不开启正常的读写带宽限制，原理与 Balancer 的 copyBlock 和 replaceBlock 操作类似。这样的话，readBlock 和 writeBlock 方法会变得更灵活，目前 readBlock、writeBlock 传入的 throttler 限流器对象为 null。

```
read = blockSender.sendBlock(out, baseStream, null);
```

这样做的好处是可以根据机器带宽资源不同，进行总带宽速率的限制。有兴趣的同学可以自己试一试。

HDFS 的 Quota 限制

Throttler 限流方案是 Hadoop 中限制资源使用的一种手段。其实在 HDFS 中，还有类似的其他限制资源滥用的方法，比如 Quota 配额机制。HDFS 中的配额机制指的是对于每个目录，我们可以设置该目录下的存储空间使用（space count）和命名空间使用

(namespace count) 计数，命名空间在此可理解为子文件数。通过配额机制我们可以很好地防止在目录下创建过多的文件或写入过量的数据。否则，就会抛出异常。相关代码的定义如下：

```
// 文件 / 目录的 Quota 计数
public class QuotaCounts {
    // 命名空间 Quota 限制与磁盘存储空间 Quota 限制
    private EnumCounters<Quota> nsSsCounts;
    // 存储类型空间的限制
    private EnumCounters<StorageType> tsCounts;
```

这里只做概括性叙述，如果有人想深入了解细节，可自行阅读相关源码。

5.2 数据平衡

对于集群运维者来说，维护集群各个节点间数据的平衡是一项基本的运维工作。如果集群内各个节点数据使用占比参差不齐，会导致集群资料利用不充分。对于数据相对多一些的节点，它上面读写的数据量以及运行任务数也会偏多，这会间接造成不同节点间负载均衡的问题，从而影响集群整体的运行效率。在 HDFS 中有一个专门的工具可以用来解决这个问题，它的名字叫做 Balancer。Balancer 工具的作用是将数据块从高数据使用量节点移动到低数据使用量节点，从而达到数据平衡的效果。本节将对此工具进行分析和讲解，主要内容包括 Balancer 工具运行的原理和 Balancer 程序的改进优化。其中 Balancer 程序的改进优化是重点内容，因为目前 Hadoop 版本中 Balancer 工具的运行效率并不是特别高，尤其在数据规模比较大的情况下，这个现象更加明显。

5.2.1 Balancer 和 Dispatcher

Balancer 和 Dispatcher 是与 HDFS Balancer 操作最紧密关联的类，Balancer 类负责找出 <source, target> 这样的起始、目标节点对，然后存入到 Dispatcher 类中，然后通过 Dispatcher 对象进行分发。不过在分发之前，会进行块的验证，判断此块是否能被移动，这里会涉及一些条件的判断，具体判断条件在后面的内容中会进行介绍。

在 Balancer 的最后阶段，会将源节点和目标节点加入到 Dispatcher 对象中，详见下面的代码：

```
/**
 * 根据源节点和目标节点，构造任务对
 */
private void matchSourceWithTargetToMove(Source source, StorageGroup target) {
    long size = Math.min(source.availableSizeToMove(), target.availableSizeToMove());
```

```

final Task task = new Task(target, size);
source.addTask(task);
target.incScheduledSize(task.getSize());
// 加入分发器中
dispatcher.add(source, target);
LOG.info("Decided to move "+StringUtil.byteDesc(size)+" bytes from "
        + source.getDisplayName() + " to " + target.getDisplayName());
}

```

Dispatcher 类中的代码进行块的分发操作:

```

private <G extends StorageGroup, C extends StorageGroup>
    void chooseStorageGroups(Collection<G> groups, Collection<C> candidates,
        Matcher matcher) {
    for(final Iterator<G> i = groups.iterator(); i.hasNext();) {
        final G g = i.next();
        for(; choose4One(g, candidates, matcher); );
        if (!g.hasSpaceForScheduling()) {
            // 如果候选节点没有空间用于调度, 则直接移除掉
            i.remove();
        }
    }
}

```

继续调用后面的方法:

```

/**
 * 从源节点列表和目标节点列表中各自选择节点组成一对, 选择顺序为同节点组、
 * 同机架, 然后是所有节点。
 */
private long chooseStorageGroups() {
    if (dispatcher.getCluster().isNodeGroupAware()) {
        // 首先匹配的条件是同节点组
        chooseStorageGroups(Matcher.SAME_NODE_GROUP);
    }

    // 然后是同机架
    chooseStorageGroups(Matcher.SAME_RACK);
    // 最后是匹配所有的节点
    chooseStorageGroups(Matcher.ANY_OTHER);

    return dispatcher.bytesToMove();
}

```

然后再通过调用 Dispatcher 的层层方法, 最后判断候选块是否合适, 处理代码如下:

```

/**
 * 决定一个块是不是一个合适的候选块的判断条件:
 * 1. 待移动的块不是正在被移动的块
 * 2. 在目标节点上没有此移动块的副本
 * 3. 移动之后, 不同机架上的块的数量应该是不变的

```

```

*/
private boolean isGoodBlockCandidate(Source source, StorageGroup target,
    DBlock block) {
    if (source.storageType != target.storageType) {
        return false;
    }

    // 如果所要移动的块是存在于正在被移动的块列表中, 则返回 false
    if (movedBlocks.contains(block.getBlock())) {
        return false;
    }
    // 如果移动的块已经存在于目标节点上, 则返回 false, 将不会予以移动
    if (block.isLocatedOn(target)) {
        return false;
    }

    // 如果开启了机架感知的配置, 则目标节点不应该有相同的块
    if (cluster.isNodeGroupAware()
        && isOnSameNodeGroupWithReplicas(target, block, source)) {
        return false;
    }

    // 需要维持机架上的块数量不变
    if (reduceNumOfRacks(source, target, block)) {
        return false;
    }
    return true;
}

```

图 5-4 为 Dispatcher 的执行过程。

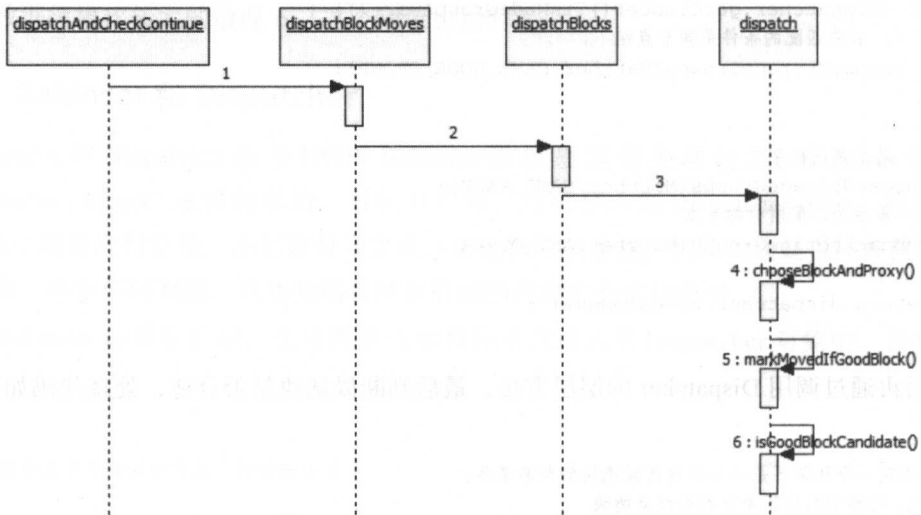


图 5-4 Dispatcher 内部流程运行图

5.2.2 数据不平衡现象

数据不平衡问题在集群规模尚且不大的时候可能不太明显，但是如果集群规模数量达到成百或上千台机器的时候，数据不平衡的问题将会越来越突出。

有两大因素可以导致节点之间出现数据不平衡的现象：

第一类情况，客户端长期写文件数据导致不均等的现象。部分机器写的的数据偏大，而部分机器写的的数据则偏小。长期积累导致了数据的不平衡。

第二类情况，新节点的上线。集群新节点部署上线时，上面存储的数据都是从零开始的，所以此时也同样需要从别的机器中同步大量的数据。

在集群规模比较大的情况下，需要平衡的数据总量也是比较大的，往往可以达到 TB 级别。数据不平衡的现象会造成拥有数据量较少的一批机器有较多远程读的操作，而 HDFS 优先的选择是大部分的数据通过本地读的方式获取。

5.2.3 Balancer 性能优化

Balancer 工具是专门用来解决上文中所提到的数据不平衡问题的，但是当需要平衡的数据达到了非常大规模的时候，当前的 Balancer 处理逻辑是否能帮助我们更快速地将数据平衡呢？是否有其他的手段帮助我们提升其性能呢？

针对第一个问题，当前的 Balancer 逻辑是否能帮助我们更快速地将数据平衡，是否存在平衡效率不高的场景呢？笔者在使用此工具的时候发现当 HDFS 中存在大量小文件的情况时，Balancer 数据平衡的效率会比较慢。再进一步分析，笔者发现平衡几个大的数据块的效率要高于平衡更多的小数据块。换句话说，大数据块在相同时间段内数据平衡效率要远高于小数据块。所以在这点上，我们可以在 Balancer 平衡程序中加入待移动块最小字节大小的限制参数，以此过滤掉一些小数据块。

其次是第二个问题，是否有其他的手段能帮助我们提升其性能？

第一个想到的办法是，加大数据平衡的带宽。DataNode 中默认的 Balancer 带宽为 10MB，在很多情况下，这个值偏小，可以通过如下命令统一调大带宽值：

```
hdfs dfsadmin -setBandwidth <bandwidth>
```

带宽值同样不能设得过大，以免影响数据的正常读写。

第二个方法，指定目标节点进行数据平衡。Balancer 程序默认是对集群中全部的节点进行数据的平衡。但有的时候，我们可以完全移除掉使用率接近于集群空间使用率的节点，专门对节点数据量少于平均值和数据量大于平均值的节点做数据平衡。这种定向平衡的方式将会提升 Balancer 的整体效率。在较新的 Balancer 版本中，所提供 `-include`、`-exclude` 参数实现了这个功能。以 `include` 参数为例子：

1) 新建 include 目标平衡节点名称列表文件，填入目标节点的 IP 或主机名。

2) 执行 `./start-balancer.sh -include -f hostfile` (节点名称列表文件的绝对路径) 命令。

第三个更加有效的办法是改造现有 Balancer 程序的代码。笔者基于 hadoop-2.7.1 版本的 Balancer 程序做了许多改造。在 Balancer 主类中新增了以下几个参数：

```
static class Parameters {
    static final Parameters DEFAULT = new Parameters(5, 1000, 1000 * 60 * 20, 1,
        BalancingPolicy.Node.INSTANCE, 10.0,
        NameNodeConnector.DEFAULT_MAX_IDLE_ITERATIONS,
        Collections.<String> emptySet(), Collections.<String> emptySet());

    final BalancingPolicy policy;
    final double threshold;
    final int maxIdleIteration;

    // 最小字节限制
    final long blockBytesNum;
    // 每次迭代最长时间限制
    final long maxIterationTime;
    // 没有可移动块情况下的最大迭代次数
    final long maxNoPendingMoveIterations;
    // 没有可移动块时的睡眠时间
    final long noPendingMoveSleepTime;
    ...
}
```

这 4 个参数分别解决了以下几方面的性能问题：

1) `blockBytesNum` 最小字节限制。这点是为了解决在上文中提到的大量小文件块造成的数据平衡效率偏低的问题。在每次筛选块的时候，额外做一次块大小的筛选判断，这需要在 `Dispatcher` 类中新增如下代码：

```
private boolean isGoodBlockCandidate(StorageGroup source, StorageGroup target,
    StorageType targetStorageType, DBlock block) {
    if (source.equals(target)) {
        return false;
    }
    if (target.storageType != targetStorageType) {
        return false;
    }

    // 检查此块是否已经被迁移过，如果是，则此块不是候选块
    if (movedBlocks.contains(block.getBlock())) {
        return false;
    }
    ...
    // 新增块大小筛选判断逻辑
    if (block.getNumBytes() < this.blockBytesNum) {
        LOG.debug("is bad block, reason: block byte num < " + this.blockBytesNum);
    }
}
```

```

    return false;
}

return true;
}

```

2) `maxIterationTime` 每次迭代最长时间限制。这里的迭代时间指的是每一轮数据平衡所花的总时间。加上这个时间是为了避免单次迭代周期所花的时间过长。

```

private void dispatchBlocks() {
    final long startTime = Time.monotonicNow();
    this.blocksToReceive = 2 * getScheduledSize();
    boolean isTimeUp = false;
    int noPendingMoveIteration = 0;
    while (!isTimeUp && getScheduledSize() > 0
        && (!srcBlocks.isEmpty() || blocksToReceive > 0)) {
        ...

        // 检测是否已经达到最大时间限制
        if (Time.monotonicNow() - startTime > Dispatcher.this.maxIterationTime) {
            LOG.info("source: " + this.getDatanodeInfo() + " is time up.");
            isTimeUp = true;
            continue;
        }

        ...
    }

    LOG.info("source: " + this.getDatanodeInfo() + " exit!");
}

```

3) `maxNoPendingMoveIterations` 没有可移动块情况下的最大迭代次数。`Balancer` 数据平衡程序会在连续 5 次没有找到可移动块的情况下退出程序，所以我们需要能够让它持久地跑下去，可以增加此参数。当然最好搭配上另外一个参数 `noPendingMoveSleepTime`，这样可以保证有足够的时间间隔。

```

private void dispatchBlocks() {
    final long startTime = Time.monotonicNow();
    this.blocksToReceive = 2 * getScheduledSize();
    boolean isTimeUp = false;
    int noPendingMoveIteration = 0;
    while (!isTimeUp && getScheduledSize() > 0
        && (!srcBlocks.isEmpty() || blocksToReceive > 0)) {
        ...
    } else {
        // 源端节点没有可移动的块时，noPendingMoveIteration 计数加 1
        noPendingMoveIteration++;
        // 进行 noPendingMoveIteration 迭代次数的判断，防止原逻辑中经过连续 5 次的
        noPendingMove 迭代后退出的现象
    }
}

```

```

        if (noPendingMoveIteration >= Dispatcher.this.maxNoPedingMoveIterations) {
            LOG.info("source: " + this.getDatanodeInfo() + " noPendingMoveIteration is
                finished." );
            resetScheduledSize();
        }
    }
    ...

```

4) noPendingMoveSleepTime 没有可移动块时的睡眠时间。这个参数指的是 Balancer 程序在没有找到可移动块前提下的缓冲时间，在以下代码中进行添加：

```

private void dispatchBlocks() {
    final long startTime = Time.monotonicNow();
    this.blocksToReceive = 2 * getScheduledSize();
    boolean isTimeUp = false;
    int noPendingMoveIteration = 0;
    while (!isTimeUp && getScheduledSize() > 0
        && (!srcBlocks.isEmpty() || blocksToReceive > 0)) {
        ...

        // 当前处于找不到块的情况时，等待一段时间
        try {
            synchronized (Dispatcher.this) {
                Dispatcher.this.wait(Dispatcher.this.noPedingMoveSleepTime); // wait for
                    targets/sources to be idle
            }
        } catch (InterruptedException ignored) {
        }
    }

    LOG.info("source: " + this.getDatanodeInfo() + " exit!");
}

```

以上 4 个参数的改造是笔者在工作中一点点去完善的。在社区上同样有相应的 JIRA 来提升 Balancer 的性能：HDFS-8825 (Enhancements to Balancer)，上文做了很多方面的优化，包括小文件块的问题，社区也做了相应的改进，同样包含在这个 JIRA 内部了。

5.3 HDFS 节点内数据平衡

上一节中我们讲述了 HDFS 节点间数据平衡的问题，在本节我们把目光移向节点内的数据平衡问题。现有 Balancer 工具并不能帮助我们平衡节点内各个磁盘间的数据，磁盘间数据的不同会造成磁盘 IO 压力的不同。在 HDFS 中，同样有类似的工具专门做这样的事情：DiskBalancer。很显然，DiskBalancer 工具操作的范围是在一个节点内。此功能目前并未发布，但是核心设计已经基本上开发完毕了，详见 JIRA：HDFS-1312 (Re-balance

disks within a Datanode)。目前社区在完善相关命令使用与文档方面的工作。想要提前使用此功能，可以自行获取 Hadoop 工程中 HDFS-1312 的分支代码。本节我们将要提前来学习与了解 DiskBalancer 相关的内容。在本节中，我们将首先讲述传统磁盘间平衡的做法，然后再引入社区方案 DiskBalancer，并对此方案进行详细的讲述。本节部分内容参考了社区 DiskBalancer 的设计文档。

5.3.1 磁盘间数据不平衡现象及问题

磁盘间数据不平衡的现象源自于长期写操作时数据大小的不均衡。因为每次写操作可以保证写磁盘的顺序性，但是无法保证每次写入的数据量都是一个规模大小。比如 A、B、C、D 四块盘，我们用默认的 RoundRobin（轮询）磁盘选择策略去写。最后四块盘的确是都写过了，但是 A、B 可能写的块较小，就 1MB，而 C、D 可能就是写满的块，128MB。

如果磁盘间数据不平衡现象确实出现了，它会给我们造成什么影响呢？有人可能会想，它不就是一个普通磁盘嘛，又不是系统盘，系统盘使用空间过高是会影响系统性能的，但是普通盘应该问题不大吧。这个观点听上去是没问题，但是只能说它考虑的太浅了。我们从 HDFS 的读写层面来对这个现象做一个分析。这里归纳出了以下两点：

第一点，磁盘间数据不平衡间接引发了磁盘 IO 压力的不同。我们都知道 HDFS 上的数据访问频率是很高的，这会涉及大量读写磁盘的操作，数据多的盘自然就会有更高频率的访问操作。如果一块盘的 IO 操作非常密集的话，势必会对它的读写性能造成影响。

第二点，高使用率磁盘导致可选存储目录减少。HDFS 在写块数据的时候，会挑选那些剩余可用空间能满足待写块大小的磁盘。如果高使用率磁盘目录过多，会导致这样的候选块变少。所以这会对 HDFS 产生影响。

5.3.2 传统的磁盘间数据不平衡解决方案

磁盘间数据不平衡现象出现了，目前我们有什么办法解决呢？下面是两种现有解决方案：

□ 方案一：节点下线再上线。将节点内数据不平衡的机器进行 Decommission 下线操作，下线之后再次上线。上线之后相当于是一个全新的节点了，数据也将会重新存储到各个盘上。这种做法给人感觉会比较粗暴，当集群规模比较小的时候代价太高，此时下线一个节点会对集群服务造成不小的影响。

□ 方案二：人工移动部分数据块存储目录。此方案相比方案一更加灵活一些，但是数据目录的移动要保证准确性，否则会造成移动完目录后数据找不到的现象。下面举一个实际的例子，比如我们想将磁盘 1 上的数据挪到磁盘 2 上。现有磁盘 1 的待移动存储目录如下：

```
/data/1/dfs/dn/ current/BP-1788246909-xx.xx.xx.xx-1412278461680/current/  
finalized/subdir0/subdir1/
```

我们移动到目标盘上的路径应该维持这样的路径格式不变，只变化磁盘所在的目录，目标路径如下：

```
/data/2/dfs/dn/current/BP-1788246909-xx.xx.xx.xx-1412278461680/current/  
finalized/subdir0/subdir1/
```

如果上述目录结构出现变化，就会造成 HDFS 找不到此数据块的情况。

5.3.3 社区解决方案：DiskBalancer

前面铺垫了这么多的内容，是为了引出本节要重点讲述的主题：DiskBalancer。DiskBalancer 从名字上可以看出，它是一个类似于 Balancer 的数据平衡工具。但是它的作用范围是被限制在了磁盘上。首先这里要说明一点，DiskBalancer 目前是未发布的功能特性，所以我们在现有发布版本中是找不到此工具的。下面笔者将会全方面介绍 DiskBalancer，让大家认识、了解这个强大的工具。

1. DiskBalancer 的设计核心

首先我们先来了解 DiskBalancer 的设计核心，这与 Balancer 有一点点的区别。Balancer 的核心点在于数据的平衡，数据平衡好就可以了。而 DiskBalancer 在设计的时候提出了两点目标：

第一点，Data Spread Report（数据分布式的汇报）。这是一个汇报的功能。也就是说，DiskBalancer 工具能支持各个节点汇报磁盘块使用情况的功能，通过这个功能集群管理者能够了解到目前集群内使用率最高的一些节点、磁盘。

第二点，Disk Balancing。第二点才是磁盘数据的平衡。但是在磁盘内数据平衡的时候，要考虑到各个磁盘存储类型的不同。之前提到过 HDFS 的异构存储，不同盘可能配置的存储类型会不同，目前 DiskBalancer 不支持跨存储类型的数据转移，所以目前都是要求在同一种存储类型下。

以上两点取自于 DiskBalancer 的设计文档（DiskBalancer 相关设计文档可见 Apache 社区 JIRA：HDFS-1312）。

2. DiskBalancer 的架构设计

此部分讨论 DiskBalancer 的架构设计。通过架构设计，我们能更好地了解它的一个整体情况。图 5-5 为 DiskBalancer 的核心架构设计。

上面过程经过了 3 个阶段，首先从 Discover（发现）到 Plan（计划），再从 Plan（计划）到 Execute（执行）。下面来详细解释这 3 个阶段。

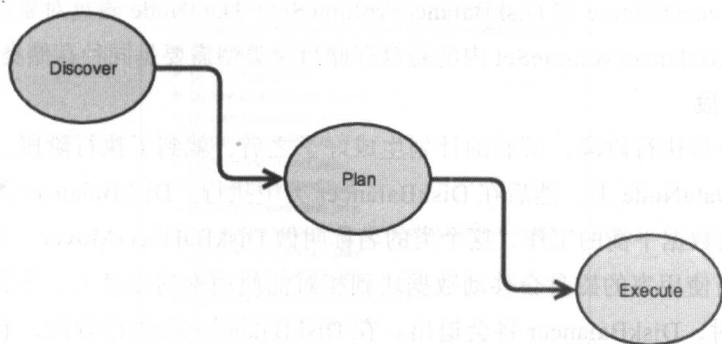


图 5-5 DiskBalancer 的架构设计图

Discover 阶段

发现阶段做的事情实际上是通过计算各个节点内的磁盘使用情况，得出需要数据平衡的磁盘列表。这里会通过 Volume Data Density（磁盘使用密度）的概念作为一个评判的标准，这个标准值将会以节点总使用率作为比较值。举个例子，如果一个节点总使用率为 75%，就是 0.75，其中 A 盘实际使用率 0.5（50%），那么 A 盘的 volumeDataDensity 密度值就等于 $0.75 - 0.5 = 0.25$ 。同理，如果超出节点总使用率的话，则密度值将会为负数。于是我们可以用节点内各个盘的 volumeDataDensity 的绝对值来判断此节点内磁盘间数据的平衡情况。如果总的绝对值的和越大，说明磁盘间数据越不平衡，这有点类似于方差的概念。Discover 阶段将会用到如下连接器对象：

❑ DBNameNodeConnector

❑ JsonConnector

❑ NullConnector

其中第一个对象会调用到 Balancer 包下的 NameNodeConnector 对象，以此来读取集群中的节点、磁盘等数据信息。

Plan 阶段

拿到上一阶段的汇报结果之后，将会进行执行计划的生成。Plan 并不是一个最小的执行单元，它的内部由各个 Step 组成。Step 中会指定好源、目标磁盘。这里的磁盘对象是一层经过包装的对象：DiskBalancerVolume，并不是原来的 FsVolume 磁盘对象。以下是 DiskBalancer 中对磁盘、节点等概念的定义：

❑ DiskBalancerCluster。通过此对象可以读取到集群中的节点信息，这里的节点信息以 DiskBalancerDataNode 的方式所呈现。

❑ DiskBalancerDataNode。此对象代表的是一个包装好后的将会进行磁盘数据平衡的 DataNode。

❑ `DiskBalancerVolume` 和 `DiskBalancerVolumeSet`。 `DataNode` 磁盘对象以及磁盘对象集合。 `DiskBalancerVolumeSet` 内的磁盘存储目录类型需要是同种存储类型。

Execute 阶段

最后一部分是执行阶段，所有的计划生成好了之后，就到了执行阶段。这些计划会被提交到各自的 `DataNode` 上，然后在 `DiskBalancer` 类中执行。 `DiskBalancer` 类中有专门的类对象来做磁盘间数据平衡的工作，这个类的名称叫做 `DiskBalancerMover`。在磁盘间数据平衡的过程中，高使用率的磁盘会移动数据块到相对低使用率的磁盘上，等到满足一定阈值关系的情况下时， `DiskBalancer` 将会退出。在 `DiskBalancer` 的执行阶段，有以下 3 项可以通过配置控制：

- ❑ 带宽的限制。 `DiskBalancer` 中同样可以支持带宽的限制，默认是 10MB，通过配置项 `dfs.disk.balancer.max.disk.throughputInMBperSec` 进行控制。
- ❑ 失败次数的限制。 `DiskBalancer` 中会存在失败次数的控制。在拷贝数据块的时候，出现 `IOException` 等异常，会进行失败次数的累加计数，如果超出最大容忍值， `DiskBalancer` 也会退出。
- ❑ 数据平衡阈值控制。 `DiskBalancer` 中可以提供一个磁盘间数据的平衡阈值，以此作为是否需要继续平衡数据的判断标准，配置项为 `dfs.disk.balancer.block.tolerance.percent`。

3. `DiskBalancer` 的代码结构

`DiskBalancer` 的相关代码最近已经合入 `hadoop-trunk` 中了，大家在 `HDFS-1312` 和 `hadoop-trunk` 中都可以进行阅读学习。笔者基于 `hadoop-trunk` 分支，对 `DiskBalancer` 进行了代码结构的分析。

目前基本上所有的 `DiskBalancer` 相关的代码都在 `org.apache.hadoop.hdfs.server.diskbalancer` 包下，在此包下又分出了 4 个子目录：

- ❑ `command`：此目录下存放 `DiskBalancer` 相关的使用命令。
- ❑ `connectors`：此目录下存放了一些用以读取节点、磁盘信息的连接器对象类。
- ❑ `datamodel`：此目录下定义了 `DiskBalancer` 中的数据实体类。
- ❑ `planner`：此目录下包含了 `plan` 计划相关类，用于在 `plan` 阶段生成计划。

图 5-6 所示为 `DiskBalancer` 的代码组成结构。

以上 4 个子目录加上 `DiskBalancer` 的主执行类和 `tool` 包下的 `DiskBalancer` 命令入口类，就构成了 `DiskBalancer` 的总代码结构。

4. `DiskBalancer` 的命令执行

`DiskBalancer` 内部提供了许多类型的命令操作，比如下面的查询命令：

```
hdfs diskbalancer -query nodename.mycluster.com
```

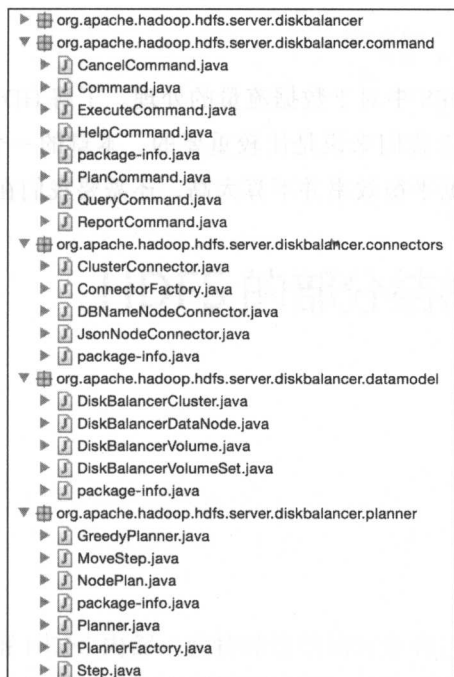


图 5-6 DiskBalancer 代码结构图

我们也可以执行相应的 plan 命令来生成 plan 计划文件：

```
hdfs diskbalancer -uri hdfs://mycluster.com -plan node1.mycluster.com
```

然后可以用生成好后的 json 格式的 plan 文件进行 DiskBalancer 的执行：

```
hdfs diskbalancer -execute /system/diskbalancer/nodename.plan.json
```

如果发现执行了错误的 plan，我们可以通过 cancel 命令进行清除：

```
hdfs diskbalancer -cancel /system/diskbalancer/nodename.plan.json
```

或

```
hdfs diskbalancer -cancel <planID> -node <nodename>
```

在 DiskBalancer 中会涉及比较多的 object-json 的关系转换，所以你会看到一些带 .json 后缀的文件。

总的来说，DiskBalancer 是一个很实用的功能特性。在 Hadoop 中，有专门的分支用于此功能的开发，分支名 HDFS-1312。感兴趣的同学可以下载 Hadoop 的最新代码进行学习。笔者非常荣幸地也向此功能提交了一个小 patch，JIRA 编号 HDFS-10560。这个功能很快就要在新版的 Hadoop 中发布了，相信会对 Hadoop 集群管理人员非常有帮助。

5.4 小结

本章主要讲述了在 HDFS 中对于数据流量的处理。了解 HDFS 内部的各个限流场景以及 Balancer 的限流原理对于我们来说是比较重要的。本章的一个难点是 Balancer 的优化，目前 HDFS 的 Balancer 数据平衡效率并不算太高，还需要我们在日后的实践和使用中进一步地优化。

HDFS 的部分结构分析

本章我们将会学习了解 HDFS 内部一些特殊的结构对象和过程的分析，了解这些对象的特殊作用或是其独特的设计。本章主要分为 3 个部分。第一节，HDFS 镜像文件的解析与反解析过程。第二节，HDFS 的数据处理过程以及数据处理中心对象 DataXceiver。第三节，HDFS 邻近信息块对象 BlockInfoContiguous，此节内容将会让我们了解到块对象是如何被组织起来的。

6.1 HDFS 镜像文件的解析与反解析

HDFS 镜像文件的解析与反解析应该拆分成“解析”过程和“反解析”过程。其中“解析”过程的意思是指把 HDFS 的镜像文件解析成用户可识别的文件形式，比如 XML 文件格式。而“反解析”过程则是“解析”步骤的逆过程，它会将 XML 格式的镜像文件重新转化为镜像文件原有的格式，即 NameNode 能够识别读取的格式。镜像文件包含着集群中所有文件元数据的信息，在很多场合下，我们需要能够读取到它的数据。通过解析与反解析的过程，用户可以更加方便地使用镜像文件。这里需要注意一点，反解析功能是社区最近完成的功能，相关 JIRA：HDFS-9835（OIV: add ReverseXML processor which reconstructs an fsimage from an XML file）。在本节中，将主要讲述 HDFS 镜像文件的解析与反解析的过程，然后介绍相关命令 `hdfs oiv` 和 `hdfs oev` 的使用。

6.1.1 HDFS 的 FsImage 镜像文件

在详细了解镜像文件的解析与反解析之前，我们需要对 FsImage 有一个初步的了解。

1. FsImage 的存储位置

没有专门运维过 Hadoop 集群的同学可能只是或多或少听到过这个名词，但是真正见到过，打开看过此文件内容的人应该不多。即便你打开了，你看到的应该是这样的乱码：

```
HDFSIMG1^V^H,<8c>ßĖ^D^Pè^G^Xô^G ^@(<8c><80><80><80>^D0P^F^H<95><80>^A^P^U1^H^B
^P<81>
<80>^A^Z^@*'^H<94>% <99>1*^Pÿÿÿÿÿÿÿÿ^?^Xÿÿÿÿÿÿÿÿ^A!í^A^A^@^@^@^@^@^@6^H^B^P<82
><80>^A^Z^Ddir0*(^Hĭ%
<99>1*^Pÿÿÿÿÿÿÿÿ^A^Xÿÿÿÿÿÿÿÿ^A!í^A^A^@^@^@^@^@^@^@<^H^A^P<83><80>^A^Z^Efile0"-
^H^A^P$%
```

可能会有人疑问为什么是这样的呢？这里笔者认为有两点原因：

第一点，因为考虑到元数据信息可能随着数据的变多而不断变大，为了缩小文件的空间大小，需要存储为二进制文件。

第二点，进行编码处理，避免直接明文保存的不安全性。

第二点原因是笔者的个人看法，主要还是第一点。镜像文件的位置信息由以下配置项所控制：

```
<property>
<name>dfs.namenode.name.dir</name>
<value></value>
</property>
```

2. FsImage 的存储信息

下面列举几个常见的可能存储的信息：

- ☐ 文件目录信息
- ☐ 位置信息
- ☐ 副本数
- ☐ 权限信息

以上几点就是传统意义上的元信息组成部分了，但是很显然，HDFS 有它自己的独特性，FsImage 保存的类型信息远远多于此，下文中还将会提到。

6.1.2 FsImage 的解析

FsImage 镜像文件的解析指的是将 HDFS 镜像文件解析成使用者能直接阅读的形式，而不是像上文中出现的乱码。这里就不得不提到一个重要的解析类：PBImageXmlWriter。解

析成 XML 文件格式只是其中一种常见的解析方式，我们同样可以直接解析展示到终端上或者以行的形式保存在普通文件中。下面是解析操作的入口：

```
Configuration conf = new Configuration();
try (PrintStream out = outputFile.equals("-") ?
    System.out : new PrintStream(outputFile, "UTF-8")) {
    // 根据传入处理器参数的不同，做不同的处理
    switch (processor) {
        case "FileDistribution":
            long maxSize = Long.parseLong(cmd.getOptionValue("maxSize", "0"));
            int step = Integer.parseInt(cmd.getOptionValue("step", "0"));
            new FileDistributionCalculator(conf, maxSize, step, out).visit(
                new RandomAccessFile(inputFile, "r"));
            break;
        case "XML":
            new PBImageXmlWriter(conf, out).visit(
                new RandomAccessFile(inputFile, "r"));
            break;
        ...
    }
}
```

进入其中 `PBImageXmlWriter` 对象的 `visit` 方法，首先会进行 `FileSummary` 的解析：

```
FileSummary summary = FSImageUtil.loadSummary(file);
```

解析 `summary` 的目的是要获取其中的 `section` 列表：

```
ArrayList<FileSummary.Section> sections = Lists.newArrayList(summary
    .getSectionsList());
```

`Section` 的概念非常的重要，每个 `Section` 会对应一类的数据。XML 格式文件也会依赖 `Section` 进行每段 XML 数据的输出。从 `FsImage` 中解析出的 `Section` 有下面几类（代码部分省略）：

```
switch (SectionName.fromString(s.getName())) {
    // 命名空间信息
    case NS_INFO:
        break;
    // 权限辅助信息
    case STRING_TABLE:
        break;
    // INode 相关信息
    case INODE:
        break;
    case INODE_REFERENCE:
        break;
    case INODE_DIR:
        break;
    // 正在构建中的文件信息
    case FILES_UNDERCONSTRUCTION:
        break;
}
```



```
// 快照相关信息
case SNAPSHOT:
    break;
case SNAPSHOT_DIFF:
    break;
// 安全管理相关的信息
case SECRET_MANAGER:
    break;
// 缓存管理相关信息
case CACHE_MANAGER:
    break;
default:
    break;
}
```

以上 10 个分支可大致分为以下 7 大类：

- ❑ 命名空间类 Section，包括 namespaceId、rollingUpgradeStartTime 等类型的变量。
- ❑ INode 相关 Section，包含了文件、目录相关 INode 的信息。
- ❑ FileUnderConstructionSection 正在构建中的文件信息。
- ❑ SnapShot 快照相关信息。
- ❑ SecretManager 安全管理相关信息。
- ❑ CacheManager 缓存管理相关信息。
- ❑ StringTable 权限相关的信息（辅助其他 Section 输出 XML 信息）。

具体 Section 里面对应有哪些信息可以在本节末尾的镜像解析完的 XML 文件样本链接中进行查看。其实 Section 本质上并不保存信息，只是提供了偏移量和长度，最终获取内部的数据还是得从镜像文件的输入流中获取，从下面的代码就可以得出这样的结论：

```
for (FileSummary.Section s : sections) {
    // 获取 Section 中的偏移量信息，并将位置定位到偏移量处
    in.getChannel().position(s.getOffset());
    // 读取此部分的输入文件内容
    InputStream is = FSImageUtil.wrapInputStreamForCompression(conf,
        summary.getCodec(), new BufferedInputStream(new LimitInputStream(
            fin, s.getLength())));

    switch (SectionName.fromString(s.getName())) {
    case NS_INFO:
        dumpNameSection(is);
    ...
}
```

我们以一个简单的解析例子来看 XML 格式的内容是怎么被输出的，以 NameSection 为例：

```
private void dumpNameSection(InputStream in) throws IOException {
```

```

// 根据给定的局部输入流, 解析成具体的 Section 对象
NameSystemSection s = NameSystemSection.parseDelimitedFrom(in);
// 根据解析好的 section 对象, 进行 XML 格式内容的输出
out.print("<" + NAME_SECTION_NAME + ">");
o(NAME_SECTION_NAMESPACE_ID, s.getNamespaceId());
o(NAME_SECTION_GENSTAMPV1, s.getGenstampV1())
    .o(NAME_SECTION_GENSTAMPV2, s.getGenstampV2())
    .o(NAME_SECTION_GENSTAMPV1_LIMIT, s.getGenstampV1Limit())
    .o(NAME_SECTION_LAST_ALLOCATED_BLOCK_ID,
        s.getLastAllocatedBlockId())
    .o(NAME_SECTION_TXID, s.getTransactionId());
out.print("</" + NAME_SECTION_NAME + ">\n");
}

```

原理很简单, 就是逐行地输出, 额外拼装成 XML 格式。到此 FsImage 文件的解析过程就是如此。

6.1.3 FsImage 的反解析

在本节开头部分已经提过, FsImage 的反解析是最近社区完成的一个新功能。这里的 FsImage 反解析指的是对解析后的 xml 文件的逆解析过程, 重新生成 FsImage 二进制文件。与以往只能保存纯粹的无法直接查阅的 FsImage 文件相比, 保存解析后的 xml 文件显然非常方便于用户的使用。反解析核心类叫做 OfflineImageReconstructor, 这个类目前需要获取 hadoop-trunk 分支代码才能看到, 目前发布版本的源码中并不存在。下面是入口处理方法:

```

try (PrintStream out = outputFile.equals("-") ?
    System.out : new PrintStream(outputFile, "UTF-8")) {
    // 根据参数传入的具体处理器的类型, 做分类处理
    switch (processor) {
        ...
        // 反解析 XML 文件过程
        case "ReverseXML":
            try {
                OfflineImageReconstructor.run(inputFile, outputFile);
            } catch (Exception e) {
                System.err.println("OfflineImageReconstructor failed: " +
                    e.getMessage());
                e.printStackTrace(System.err);
                System.exit(1);
            }
            break;
    }
}

```

接着进入 OfflineImageReconstructor 的 run 方法:

```

public static void run(String inputPath, String outputPath)
    throws Exception {
    MessageDigest digester = MD5Hash.getDigester();
}

```

```

FileOutputStream fout = null;
File fouthash = new File(outputPath + ".md5");
Files.deleteIfExists(fouthash.toPath()); // delete any .md5 file that exists
CountingOutputStream out = null;
FileInputStream fis = null;
InputStreamReader reader = null;
try {
    // 如果目标输出路径已存在，则进行删除
    Files.deleteIfExists(Paths.get(outputPath));
    fout = new FileOutputStream(outputPath);
    // 根据输入路径得到输入流对象
    fis = new FileInputStream(inputPath);
    reader = new InputStreamReader(fis, Charset.forName("UTF-8"));
    out = new CountingOutputStream(
        new DigestOutputStream(
            new BufferedOutputStream(fout), digester));
    OfflineImageReconstructor oir =
        new OfflineImageReconstructor(out, reader);
    // 进行镜像文件反解析的过程
    oir.processXml();
} finally {
    IOUtils.cleanup(LOG, reader, fis, out, fout);
}
// Write the md5 file
MD5FileUtils.saveMD5File(new File(outputPath),
    new MD5Hash(digester.digest()));
}

```

以上代码主要做了以下几件事情：

- 先判断之前是否已经存在目标文件，如果存在则进行删除。
- 在内部进行了 `OfflineImageReconstructor` 对象的构建。
- 然后进行 XML 文件的处理。

然后进入 `OfflineImageReconstructor` 的构造函数，里面会做哪些初始化操作呢？代码如下：

```

private OfflineImageReconstructor(CountingOutputStream out,
    InputStreamReader reader) throws XMLStreamException {
    this.out = out;
    XMLInputFactory factory = XMLInputFactory.newInstance();
    this.events = factory.createXMLEventReader(reader);
    this.sections = new HashMap<>();
    this.sections.put(NameSectionProcessor.NAME, new NameSectionProcessor());
    this.sections.put(INodeSectionProcessor.NAME, new INodeSectionProcessor());
    this.sections.put(SecretManagerSectionProcessor.NAME,
        new SecretManagerSectionProcessor());
    this.sections.put(CacheManagerSectionProcessor.NAME,
        new CacheManagerSectionProcessor());
    this.sections.put(SnapshotDiffSectionProcessor.NAME,

```

```

        new SnapshotDiffSectionProcessor();
this.sections.put(INodeReferenceSectionProcessor.NAME,
    new INodeReferenceSectionProcessor());
this.sections.put(INodeDirectorySectionProcessor.NAME,
    new INodeDirectorySectionProcessor());
this.sections.put(FilesUnderConstructionSectionProcessor.NAME,
    new FilesUnderConstructionSectionProcessor());
this.sections.put(SnapshotSectionProcessor.NAME,
    new SnapshotSectionProcessor());
this.isoDateFormat = PBImageXmlWriter.createSimpleDateFormat();
}

```

在这里出现了很多的 SectionProcessor 对象，这是对应于 PBImageXmlWriter 类中的 9 大 Section 的。但是好像 stringTable 没有对应到，在下文中将会提到。

```

case STRING_TABLE:
    loadStringTable(is);
    break;

```

那么这些 SectionProcessor 对象是如何解析 XML，然后将信息写入 FsImage 的输出文件的呢？以 NameSectionProcessor 为例：

```

private class NameSectionProcessor implements SectionProcessor {
    static final String NAME = "NameSection";

    @Override
    public void process() throws IOException {
        Node node = new Node();
        loadNodeChildren(node, "NameSection fields");
        NameSystemSection.Builder b = NameSystemSection.newBuilder();
        Integer namespaceId = node.removeChildInt(NAME_SECTION_NAMESPACE_ID);
        if (namespaceId == null) {
            throw new IOException("<NameSection> is missing <namespaceId>");
        }
        // 将相关的值设入 Section 的 Builder 对象中
        b.setNamespaceId(namespaceId);
        Long lval = node.removeChildLong(NAME_SECTION_GENSTAMPV1);
        if (lval != null) {
            b.setGenstampV1(lval);
        }
        ...
        node.verifyNoRemainingKeys("NameSection");
        // 用 Section 的 Builder 对象来创建 Section
        NameSystemSection s = b.build();
        if (LOG.isDebugEnabled()) {
            LOG.debug(SectionName.NS_INFO.name() + " writing header: {" +
                TextFormat.printToString(s) + "}");
        }
        // 将 Section 对象内容写出到输出流文件中
        s.writeDelimitedTo(out);
    }
}

```

```

        recordSectionLength(SectionName.NS_INFO.name());
    }
}

```

以上的过程可以归纳为如下步骤：

1) 从 XML 文件中解析数据加载到 Node 节点对象：

```
loadNodeChildren(node, "NameSection fields");
```

2) 从 Node 对象中通过 key 名称获取对应的值并移除原有 key-value 对：

```
Long lval = node.removeChildLong(NAME_SECTION_GENSTAMPV1);
```

3) 检查 Node 对象是否已经没有剩余的 key：

```
node.verifyNoRemainingKeys("NameSection");
```

4) 将解析好后的信息写入目标镜像文件：

```
s.writeDelimitedTo(out);
```

其他 SectionProcessor 的处理与此类似，这里就不进行多余的介绍了。回到刚刚提到的问题，为什么 stringTable 没有对应的 processor 呢？因为在生成的 FsImage 中会重新进行构造：

```

int registerStringId(String str) throws IOException {
    Integer id = stringTable.get(str);
    if (id != null) {
        return id;
    }
    int latestId = latestStringId;
    if (latestId >= 0xffffffff) {
        throw new IOException("Cannot have more than 2**25 " +
            "strings in the fsimage, because of the limitation on " +
            "the size of string table IDs.");
    }
    // 将 id 值加入 stringTable 中
    stringTable.put(str, latestId);
    latestStringId++;
    return latestId;
}

```

在与权限相关的操作中会调用到此方法：

```

private long permissionXmlToU64(String perm) throws IOException {
    String components[] = perm.split(":");
    if (components.length != 3) {
        throw new IOException("Unable to parse permission string " + perm +
            ": expected 3 components, but only had " + components.length);
    }
    String userName = components[0];

```

```

String groupName = components[1];
String modeString = components[2];
// 将以上名称写入 stringTable
long userNameId = registerStringId(userName);
long groupNameId = registerStringId(groupName);
long mode = new FsPermission(modeString).toShort();
return (userNameId << 40) | (groupNameId << 16) | mode;
}

```

在 `OfflineImageReconstructor` 中的 `processXml` 方法中, 采取了循环逐个解析的方式进行 `Section` 的处理:

```

private void processXml() throws Exception {
    LOG.debug("Loading <fsimage>.");
    expectTag("fsimage", false);
    // 读入镜像文件的 version 版本号
    readVersion();
    // 在镜像文件中开始写入 magic 数字信息
    out.write(FSImageUtil.MAGIC_HEADER);
    // 准备写出一系列的 Section 信息
    sectionStartOffset = FSImageUtil.MAGIC_HEADER.length;
    final HashSet<String> unprocessedSections =
        new HashSet<>(sections.keySet());
    // 循环遍历处理 Section, 直到 Section 都处理完毕
    while (!unprocessedSections.isEmpty()) {
        XMLEvent ev = expectTag("[section header]", true);
        ....
        SectionProcessor sectionProcessor = sections.get(sectionName);
        if (sectionProcessor == null) {
            throw new IOException("Unknown FSImage section " + sectionName +
                ". Valid section names are [" +
                StringUtils.join(", ", sections.keySet()) + "]);");
        }
        unprocessedSections.remove(sectionName);
        sectionProcessor.process();
    }

    // 写出 stringTable 的 Section 信息
    writeStringTableSection();
    ...
}

```

while 循环结束之后, 最后会把 `stringTable` 再次写入。

`FsImage` 的解析与反解析的过程具有非常强的对称性。图 6-1 所示为 `FsImage` 的解析与反解析流程。

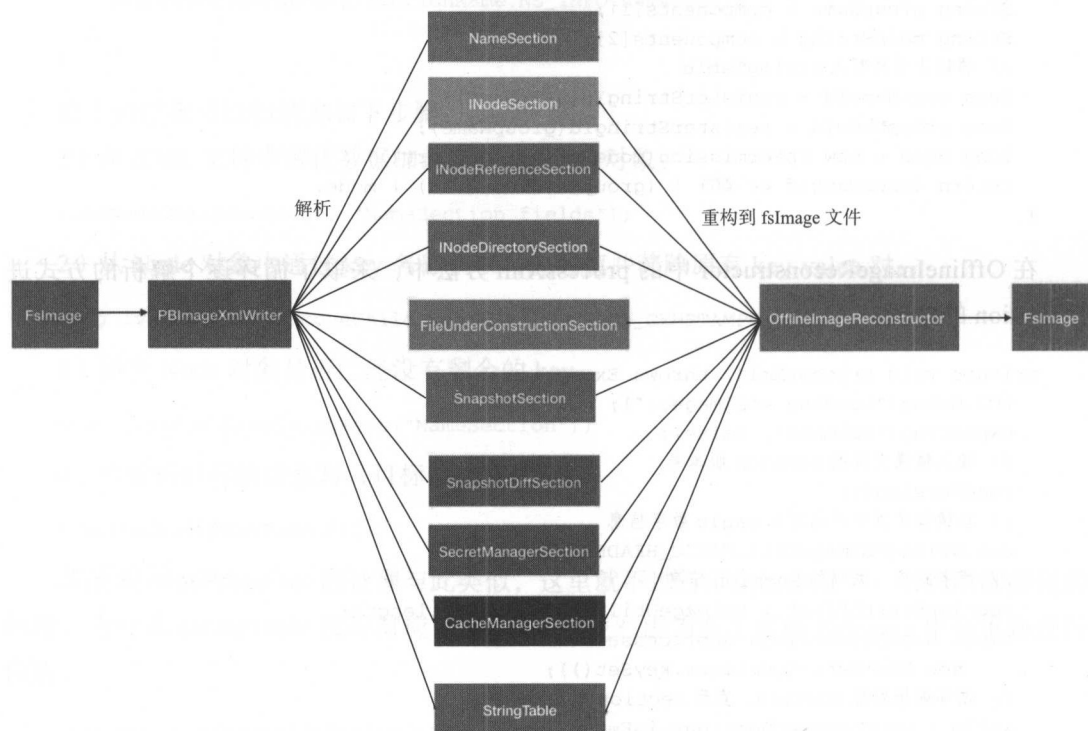


图 6-1 FsImage 的解析与反解析流程图

6.1.4 HDFS 镜像文件的解析与反解析命令

HDFS 关于镜像解析的命令主要以 `hdfs oiv` 打头，`oiv` 为 `OfflineImageView` 的缩写。解析与反解析在这里的区别只是处理器的不同。目前总共有以下 5 大处理器：

- ❑ XML
- ❑ ReverseXML
- ❑ FileDistribution
- ❑ Web
- ❑ Delimited

调用命令如下：

```
Usage: bin/hdfs oiv -p XML/ReverseXML -i INPUTFILE -o OUTPUTFILE
```

`inputFile` 和 `outputFile` 为必填参数。如果想看其他参数，可以输入以下命令进行查阅：

```
bin/hdfs oiv -h
```

其中，`hdfs oiv` 命令除了上面提到的将镜像文件解析为可读的 XML 文件之外，还有

一个很实用的分析文件目录的功能。它能根据镜像文件，离线计算出 HDFS 中所有文件和目录数，并且能够对文件做大小分类，统计出在各个大小区间内的文件数量，借此可以判断集群中是否存在大量的小文件。文件大小分析功能需要用 `-p FileDistribution` 的处理器，我们还可以根据自己的需要限定一个区间大小 `step` 和最大上限值大小 `maxSize`，使用命令如下：

```
bin/hdfs oiv -p FileDistribution -i INPUTFILE -o OUTPUTFILE
```

使用此命令，可以免去用户额外编写 java 程序去做 HDFS 文件扫描处理的过程。通过用户程序的方式去分析 HDFS 文件目录情况会造成对 NameNode 大量文件信息的请求。另外笔者在工作中使用此工具时发现了两个比较影响用户体验的问题：

- ❑ 解析镜像文件有时会出现数组越界，继而导致分析程序退出现象。后来经过调试程序，笔者发现这样的情况发生在 `maxSize` 无法完全整除 `step` 值的特殊情况，随后笔者向社区提交了 issue，并进行了解决，JIRA 编号 HDFS-10691。
- ❑ `FileDistribution` 处理器解析结果可读性较差，无法让用户直观地理解它的意思。于是笔者对其进行了优化，在 `FileDistribution` 处理器中新增了 `-format` 参数来可视化输出的结果，详细内容可见 JIRA：HDFS-10778。

用 hdfs oev 命令分析 editlog 文件

与 `hdfs oiv` 命令类似，HDFS 同样提供了分析 editlog 日志文件的工具命令：`hdfs oev`。`oev` 是 `OfflineEditsViewer` 的缩写，在 `hdfs oev` 命令中，可以分析出指定 editlog 文件中各个操作记录类型以及相应的计数值，详细的命令使用方法可以输入 `hdfs oev` 进行查看。

希望本节内容能够对大家有所帮助，对 `FsImage` 以及 `hdfs oiv` 有更多了解。在学习 `OfflineImageReconstructor` 源码的同时，笔者发现了其中部分可以优化的地方，向社区提交了 JIRA：HDFS-9951（`Use string constants for XML tags in OfflineImageReconstructor`），目前已被社区接受。有兴趣的同学，可以查看此 JIRA。最后附上一个镜像文件解析好的 XML 格式文件样例（可能部分人并没有看过 HDFS 的镜像文件里面到底保存了什么样的内容），方便大家对照源码学习。

镜像文件的 XML 解析文件样本链接如下：<https://github.com/linyiquan/open-source-patch/blob/master/hdfs/others/hdfs-fsimage/fsimageSample.xml>。

6.2 DataNode 数据处理中心 DataXceiver

在 `DataNode` 中，包含着一个数据处理中心类：`DataXceiver`。数据读写的所有操作都会

经过此类。用简单的一句话来概括它的作用：DataXceiver 个数的多少，在一定程度上能反映出此节点的忙碌程度。本节将为大家介绍 DataXceiver 内部的过程调用，以及普通的文件读写操作是如何在 DataXceiver 中调用的。

我们从大的层面往小说，你就知道它有多重要了。我们使用 Hadoop 系统，最看重的是两个字：存储。存储的过程中，什么又是最看重的呢？那当然是数据了。而这些数据都是存在于各个 DataNode 之上的。所以掌控 DataNode 读写操作的服务就显得尤为重要了。而这个控制中心就在 DataXceiver 中。

6.2.1 DataXceiver 的定义和结构

DataXceiver 是干什么用的呢，很多人只知 DataNode，而不知另外一个很重要的线程服务 DataXceiver。在 HDFS 中对于 DataXceiver 的解释如下：

```
// 处理输入、输出数据流的线程类
class DataXceiver extends Receiver implements Runnable {
    ...
}
```

笔者的个人理解是数据流的处理中心。DataXceiver 线程数的多少在一定程度上能反映出一个节点的忙碌程度。DataXceiver 这个类中包含的变量和方法还是比较多的，这里不大建议读者逐行详细地去阅读内部的代码。我们去学习一个机制、原理的时候，主要要明白的是结构。比如我们现在要去学习 DataXceiver 这个类，我们的目标是去了解这个类主要做了哪些操作，上游被哪些对象调用，下游又调用了哪些类，具体的代码细节等碰到具体的问题时再去分析即可，否则可能会被里面复杂的逻辑绕晕。毕竟 Hadoop 是一个成熟的分布式程序，不是一时半会能够立刻理解的。

为了更好地理解这个“数据处理中心”，我们需要去了解这个类的整体结构，在此之前不妨浏览一下其中的内部方法。图 6-2 所示为 DataXceiver 内部的处理方法。

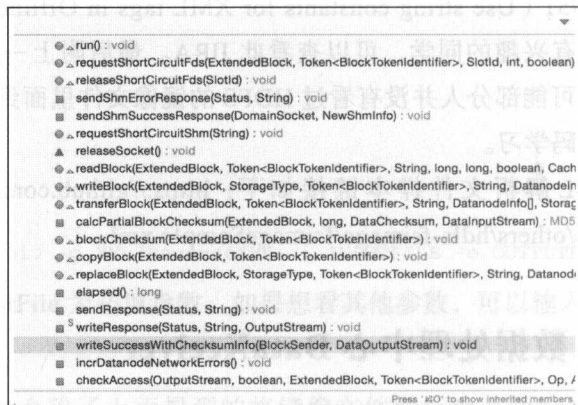


图 6-2 DataXceiver 内部方法操作

首先，这是一个线程服务，执行入口是 `run` 方法。执行 `run` 方法。我们可以找到与之关联的操作：

```
public void run() {
    int opsProcessed = 0;
    Op op = null;

    ...

    // 在下面的循环中周期性处理请求
    do {
        updateCurrentThreadName("Waiting for operation #" + (opsProcessed + 1));

        try {
            if (opsProcessed != 0) {
                assert dnConf.socketKeepaliveTimeout > 0;
                peer.setReadTimeout(dnConf.socketKeepaliveTimeout);
            } else {
                peer.setReadTimeout(dnConf.socketTimeout);
            }
            // 读取请求操作码
            op = readOp();
        } catch (InterruptedException ignored) {
            // 等待客户端 RPC 请求超时
            break;
        } catch (IOException err) {
            // EOFException 异常处理
            if (opsProcessed > 0 &&
                (err instanceof EOFException || err instanceof ClosedChannelException)) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Cached " + peer + " closing after " + opsProcessed + " ops");zhu
                }
            } else {
                // 增加网络 IO 异常计数
                incrDatanodeNetworkErrors();
                throw err;
            }
            break;
        }
    }

    // 重新设置超时时间
    if (opsProcessed != 0) {
        peer.setReadTimeout(dnConf.socketTimeout);
    }

    opStartTime = monotonicNow();
    // 处理请求操作码
    processOp(op);
    ++opsProcessed;
} while ((peer != null) &&
```

```

        (!peer.isClosed() && dnConf.socketKeepaliveTimeout > 0));
        ...

```

在 run 方法中间的主循环方法中，可以看到一次 readOp 操作，对应的是一次 processOp 操作。Op 的意思是操作码，readOp 方法会从输入流中读取操作码：

```

// 读取 Op 操作码方法
protected final Op readOp() throws IOException {
    final short version = in.readShort();
    if (version != DataTransferProtocol.DATA_TRANSFER_VERSION) {
        throw new IOException( "Version Mismatch (Expected: " +
            DataTransferProtocol.DATA_TRANSFER_VERSION +
            ", Received: " + version + " )");
    }
    // 从输入流中读取操作码
    return Op.read(in);
}

```

而 processOp 方法则会进行操作码的判断处理，主要分为以下几类操作码的判断：

```

// process Op 操作码
protected final void processOp(Op op) throws IOException {
    switch(op) {
        // 读取块的处理
        case READ_BLOCK:
            opReadBlock();
            break;
        // 写块的处理
        case WRITE_BLOCK:
            opWriteBlock(in);
            break;
        // 取代块的处理，Balancer 数据平衡的时候会调用到
        case REPLACE_BLOCK:
            opReplaceBlock(in);
            break;
        // 复制块的处理，Balancer 数据平衡的时候会调用到
        case COPY_BLOCK:
            opCopyBlock(in);
            break;
        // 读取校验和的处理
        case BLOCK_CHECKSUM:
            opBlockChecksum(in);
            break;
        // 传输数据块的处理
        case TRANSFER_BLOCK:
            opTransferBlock(in);
            break;
        // 以下 3 类处理均与 ShortCircuit (短路读) 相关
        case REQUEST_SHORT_CIRCUIT_FDS:
            opRequestShortCircuitFds(in);

```

```

        break;
    case RELEASE_SHORT_CIRCUIT_FDS:
        opReleaseShortCircuitFds(in);
        break;
    case REQUEST_SHORT_CIRCUIT_SHM:
        opRequestShortCircuitShm(in);
        break;
    default:
        throw new IOException("Unknown op " + op + " in data stream");
    }
}

```

总共 9 种类型，对应着 9 种处理方法。到此，DataXceiver 的基本结构慢慢清晰了，如图 6-3 所示。

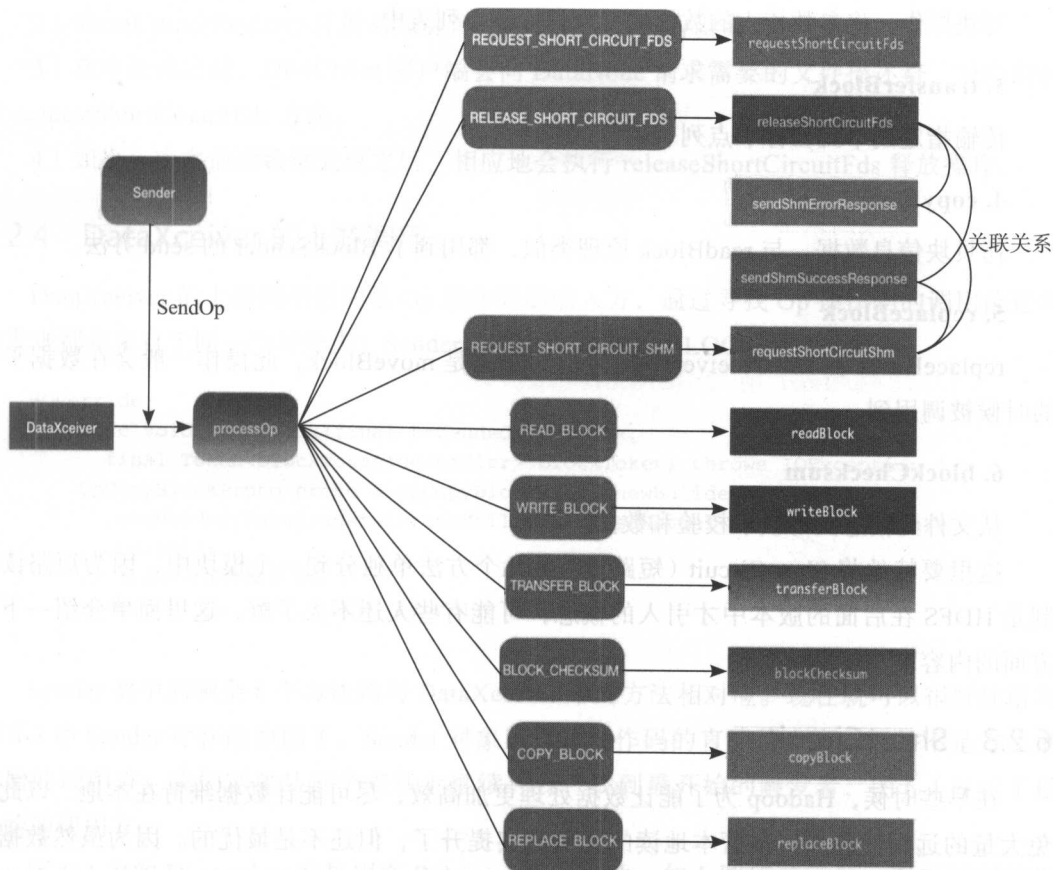


图 6-3 DataXceiver 的基本处理结构

左上方的 Sender 对象在下文中会具体讲述，此处可以先忽略。

6.2.2 DataXceiver 下游处理方法

从上文中的结构图中我们看到了处理操作码的 9 个方法和 2 个回复方法。这 9 个方法可以大致分为两大类。第一类，普通读写块的操作方法。划分到普通读写块方法的有 `readBlock`、`writeBlock`、`transferBlock`、`copyBlock`、`replaceBlock` 和 `blockChecksum`，剩下的一类方法则属于 `ShortCircuit`（短路读）相关的方法。下面对这些方法做场景分析：

1. readBlock

方法名已经表明了此方法的操作了，自然是读取块信息的操作，一般用于远程读或者本地读操作。

2. writeBlock

写块操作，将参数传入的数据块写入目标节点列表中。

3. transferBlock

传输指定副本到目标节点列表中。

4. copyBlock

拷贝块信息数据，与 `readBlock` 原理类似，都用到了 `BlockSender` 的 `send` 方法。

5. replaceBlock

`replaceBlock` 在 `DataXceiver` 中更接近的意思是 `moveBlock`，此操作一般会在数据平衡的时候被调用到。

6. blockChecksum

从文件元信息头部读取校验和数据。

这里要特地将 `ShortCircuit`（短路读）的几个方法单独分到一个模块中，因为短路读机制是 HDFS 在后面的版本中才引入的概念，可能有些人还不大了解，这里简单介绍一下这方面的内容。

6.2.3 ShortCircuit

在早些时候，Hadoop 为了能让数据处理更加高效，尽可能让数据维持在本地，以此避免大量的远程读操作。尽管本地读的比例确实提升了，但还不是最优的。因为虽然数据是在本地，但是每次客户端读取数据，还是需要走 `DataNode` 这一层，在其间还是会走网络通信的模块，能不能以类似于直接读取本地文件系统的方式去读本地的数据呢？`ShortCircuit` 就是源自于这个想法而诞生的。

1. ShortCircuit 本地读的实现

HDFS 采用了 Linux 操作系统中的 Unix Domain Socket 技术来实现 ShortCircuit 功能。ShortCircuit 是一种进程间通信的方式，很重要的一个特性是可以在进程间传递文件描述符，借此来进行进程间的通信，从而实现了本地读。关于 ShortCircuit 本地读更多细节的文章可以阅读 cloudera 官网上的一篇文章：“How Improved Short-Circuit Local Reads Bring Better Performance and Security to Hadoop”。

2. ShortCircuit 机制

在 HDFS 中用的是共享内存片段（short-circuit memory segments）来实现数据的读操作。DFSClnt 客户端通过 ShortCircuit 机制实现本地读的简要过程如下：

- 1) DFSClnt 客户端从 DataNode 请求共享内存片段。
- 2) ShortCircuitRegistry 注册对象会产生并管理这些内存对象。
- 3) 在本地读之前，DFSClnt 客户端会向 DataNode 请求需要的文件描述符，对应的就是 requestShortCircuitFds 方法。
- 4) 如果一次本地读数据完成之后，相应地会执行 releaseShortCircuitFds 释放操作。

6.2.4 DataXceiver 的上游调用

DataXceiver 的上游调用指的是 Op 操作码的输入方，通过寻找 Op 操作码的调用位置可以发现都是来自于同一个对象类：Sender。其中以 COPY_BLOCK 操作码为例：

```
@Override
public void copyBlock(final ExtendedBlock blk,
    final Token<BlockTokenIdentifier> blockToken) throws IOException {
    OpCopyBlockProto proto = OpCopyBlockProto.newBuilder()
        .setHeader(DataTransferProtoUtil.buildBaseHeader(blk, blockToken))
        .build();
    // 发送拷贝块的操作请求
    send(out, Op.COPY_BLOCK, proto);
}
```

Sender 类中的剩余 8 个方法均与 DataXceiver 中的方法相对应。现在就可以很好地解释图 6-3 中 Sender 存在的原因了。Sender 对象虽然是操作码的直接传入类，但并不是方法最初的调用方，我们需要从这个点往上继续寻找，找到最开始的触发者，图 6-4 显示了最初的调用方。

图 6-4 中的 Dispatcher 类是用在 Balancer 操作中的。如上图所显示的，真正读写数据的发起方是我们经常碰到的 DFSClnt、DFSOutputStream 和 BlockReader 这些对象类。至此，DataXceiver 的上游调用以及下游处理就完全打通了。

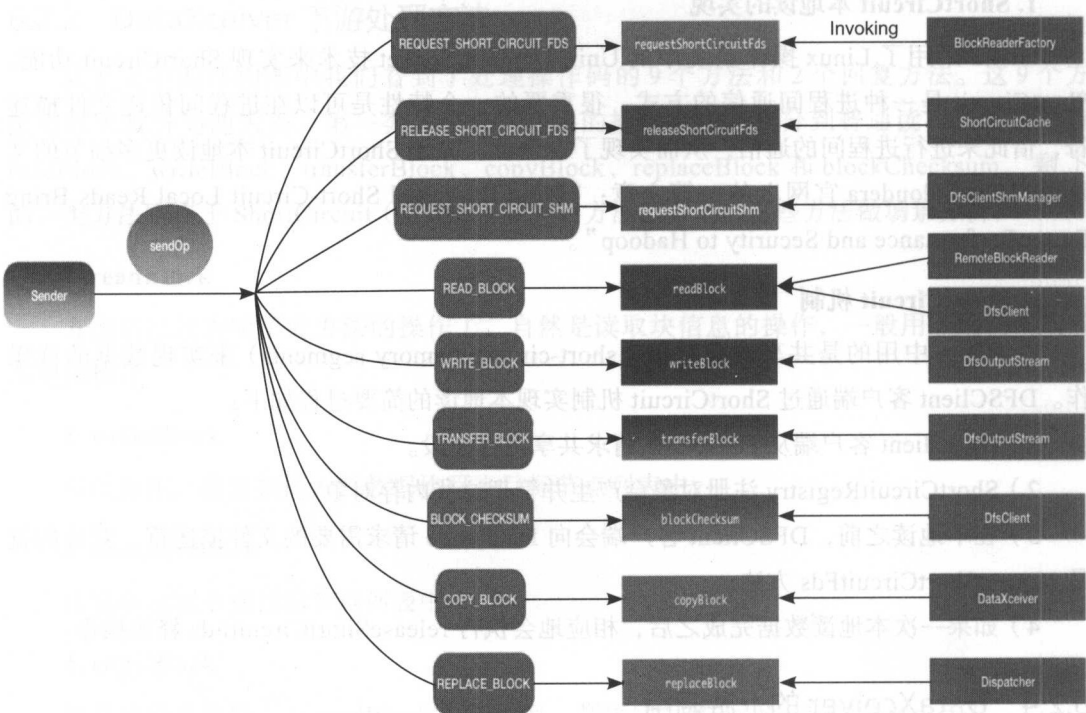


图 6-4 Sender 与 DataXceiver 的关系图

6.2.5 DataXceiver 与 DataXceiverServer

提到 DataXceiver 就不得不提 DataXceiverServer。DataXceiverServer 会保存每次新启动的 DataXceiver 线程。在它的主循环方法中，会进行 DataXceiver 的创建：

```
@Override
public void run() {
    Peer peer = null;

    while (datanode.shouldRun && !datanode.shutdownForUpgrade) {
        try {
            peer = peerServer.accept();

            // Make sure the xceiver count is not exceeded
            // 获取 DataNode 目前已有的 DataXceiver 个数，保证 DataXceiver 个数不超过上限
            int curXceiverCount = datanode.getXceiverCount();
            if (curXceiverCount > maxXceiverCount) {
                throw new IOException("Xceiver count " + curXceiverCount
                    + " exceeds the limit of concurrent xcievers: "
                    + maxXceiverCount);
            }
            // 启动新的 DataXceiver 线程服务
```



```

        new Daemon(datanode.threadGroup,
            DataXceiver.create(peer, datanode, this))
            .start();
    } catch (SocketTimeoutException ignored) {

```

随后 DataXceiver 会加入 DataXceiverServer 的 map 对象中:

```

public void run() {
    int opsProcessed = 0;
    Op op = null;

    try {
        dataXceiverServer.addPeer(peer, Thread.currentThread(), this);
        ...
        synchronized void addPeer(Peer peer, Thread t, DataXceiver xceiver)
            throws IOException {
            if (closed) {
                throw new IOException("Server closed.");
            }
            // 将 DataXceiver 对象加入 map 中进行保存
            peers.put(peer, t);
            peersXceiver.put(peer, xceiver);
        }
    }
}

```

所以 DataXceiverServer 与 DataXceiver 的关系可以说是包含与被包含的关系。

6.3 HDFS 邻近信息块: BlockInfoContiguous

在 HDFS 中,数据的存储是以块的形式存在的。而每个块的默认副本数是 3 个,于是在 HDFS 中会存在 3 个相同的块副本分布在不同的 DataNode 节点上。这些相同块的副本信息由 HDFS 邻近信息块 (BlockInfoContiguous) 所包含。在 BlockInfoContiguous 类中,这些副本块信息被巧妙地组织,以此节省出更多的内存空间。本节我们将要全面地学习 BlockInfoContiguous 类,了解它的作用、设计原理以及其内部各式各样的链表操作。

在最新的 hadoop-trunk 的代码中这个类已经做了比较大的改动,所以这里笔者要说明一下本节所使用的源码版本是 hadoop-2.7.1。在此版本中,BlockInfoContiguous 类用来寻找副本块的直接信息类,它的内部包含了针对某个块的所有副本块位置信息,同时它也包含了自身块的详细信息。在官方的源码中对 BlockInfoContiguous 的解释如下:

```

// 此类包含了给定块的具体信息,同时包含了此块其余副本的位置信息
@InterfaceAudience.Private
public class BlockInfoContiguous extends Block
    implements LightweightGSet.LinkedElement {

```

在 BlockInfoContiguous 类中,有两个关键的内部对象: BlockCollection 和 triplets。前

者保存了类似副本数、副本位置等一些信息，而 triplets 对象数组则是本节的一个重点。所以下面详细地分析 triplets 对象数组的设计结构和思想。

6.3.1 triplets 对象数组

triplets 对象初始化的时候是一个 Object 对象数组，但是在赋值的时候，会存储两类对象。以下是 triplets 数组变量声明：

```
private Object[] triplets;
```

通过官方源码的注释，我们可以归纳出下面几点信息：

- 对于当前块的信息，块存在于多个节点位置中，假如存储于 i 个节点，则 triplets 对象数组大小就是 $3 * i$ 个，一般存储的节点数视副本系数而定。
- 对 triplets 每 3 个为一单位的数组来说， $triplets[3 * i]$ 保存的是节点位置信息， $triplets[3 * i + 1]$ 保存的是此节点位置中前一个块对象的信息， $triplets[3 * i + 2]$ 保存的则是后一个块对象的信息，而保存块信息对象的类同样是 BlockInfoContiguous。

所以我们可以稍稍地想象一下，这其实是一个“巨大的链表”。但是 HDFS 为了更高效地使用内存，并没有采用 JDK 自带的像 LinkedList 这样的链表结构。BlockInfoContiguous 内部结构如图 6-5 所示。

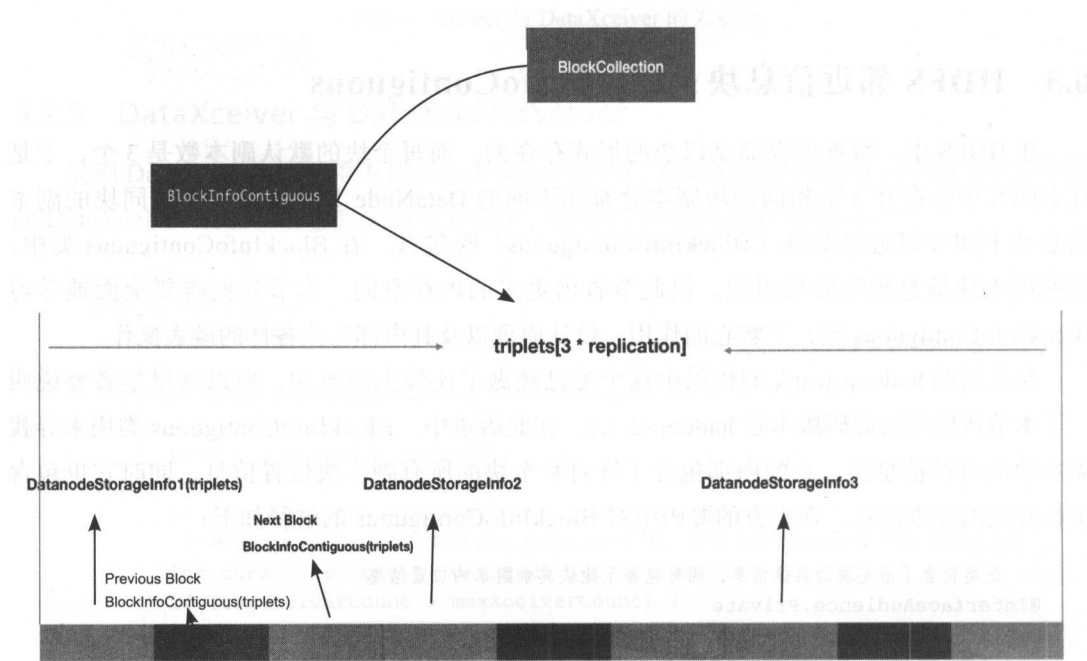


图 6-5 BlockInfoContiguous 结构图

DatanodeStorageInfo1、2、3 是当前块存储的节点，所以 triplets 的长度是根据副本数进行初始化的：

```
// 构造一个实例对象
public BlockInfoContiguous(short replication) {
    this.triplets = new Object[3*replication];
    this.bc = null;
}
```

每个节点上会存储大量的块，通过块的下一个块或前一个块，我们可以遍历完节点上的所有块。在每个 DataNodeStorageInfo 中，所持有块的关联结构如图 6-6 所示。

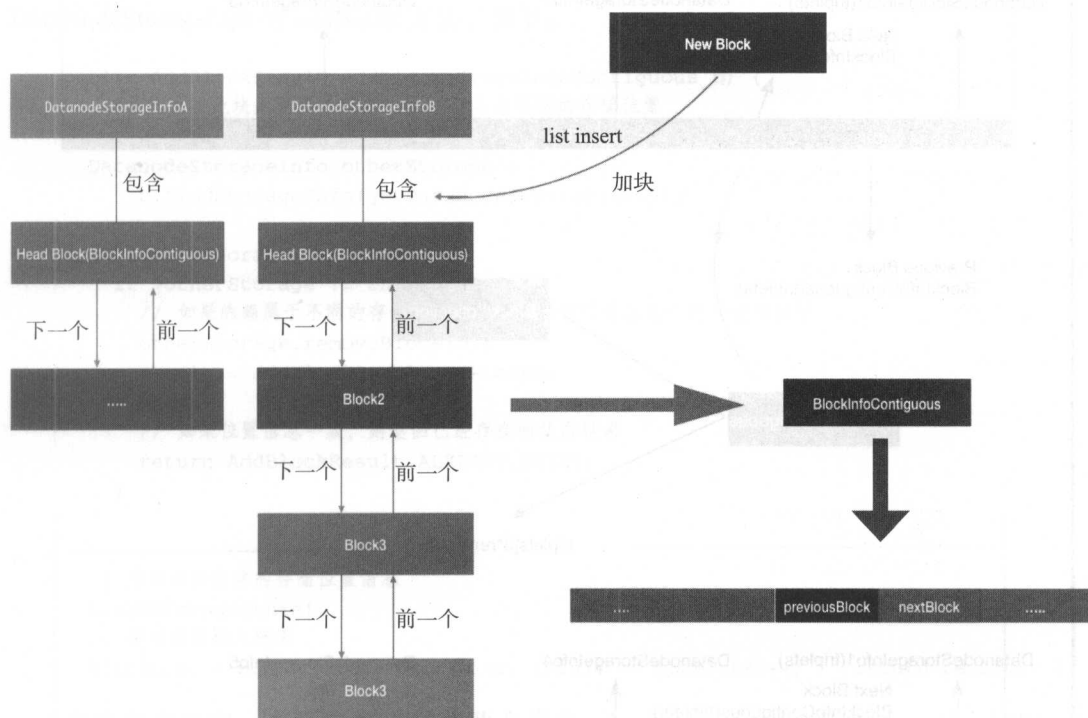


图 6-6 DataNodeStorageInfo 上的块关联图

这里的 head 头块，对应的是 DataNodeStorageInfo 中的 blocklist 对象：

```
private volatile BlockInfoContiguous blockList = null;
```

图 6-6 中同一个节点中的块与块之间的连接关系如图 6-7 所示。

DataNode 上关于块的操作都会在它所维护的块列表中进行操作。

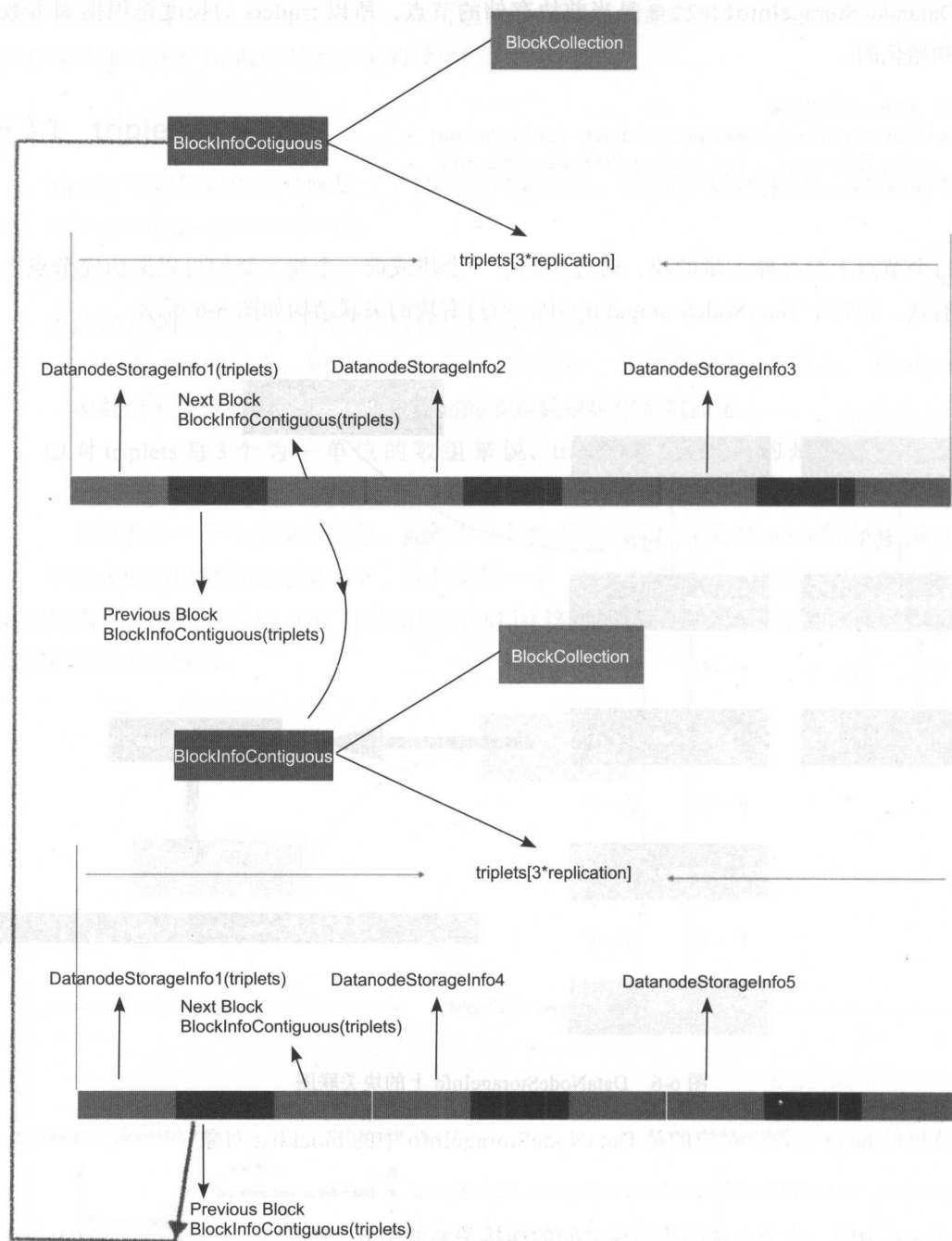


图 6-7 节点内的块关系图

6.3.2 BlockInfoContiguous 的链表操作

DataNode 上的块的添加删除动作对照过来就是 BlockInfoContiguous 的链表操作。其中的操作主要分为两类：一个是 addBlock 添加块的操作，另外一个 removeBlock 移除块的操作。这两个方法都定义在 DataNodeStorageInfo 类中，最终映射到块的链表操作方法是 listInsert 和 listRemove。此外在 BlockInfoContiguous 类中，还有一个 moveBlockToHead 操作，此操作的作用是将块移到链表头部。下面主要分析一下这 3 个方法。

1. listInsert

listInsert 的操作效果是往对应节点链表中添加一个块，触发此操作的原始方法为 DataNodeStorageInfo 的 addBlock 方法，如下：

```
public AddBlockResult addBlock(BlockInfoContiguous b) {
    // 首先检查此块是否属于同个 DataNode 上不同的存储位置
    AddBlockResult result = AddBlockResult.ADDED;
    DataNodeStorageInfo otherStorage =
        b.findStorageInfo(getDatanodeDescriptor());

    if (otherStorage != null) {
        if (otherStorage != this) {
            // 如果的确属于不同的存储位置，则在之前的位置信息中将当前块移除
            otherStorage.removeBlock(b);
            result = AddBlockResult.REPLACED;
        } else {
            // 如果位置信息一致，则返回已经存在的状态结果
            return AddBlockResult.ALREADY_EXIST;
        }
    }

    // 为块添加当前的存储位置信息
    b.addStorage(this);
    // 将当前块插入链表
    blockList = b.listInsert(blockList, this);    numBlocks++;    return result; }
}
```

在这个方法中，主要关注末尾的两个操作：b.addStorage 和 b.listInsert。b.addStorage 的意思是在新增的块中赋值当前的节点信息，因为此块是被写入当前节点中的，要把节点信息写入块自身维护的链表信息中。

```
// 为块增加一个存储位置信息
boolean addStorage(DataNodeStorageInfo storage) {
    // triplets 数组扩容 1 个单位的节点位置，相当于扩充 3 个对象
    int lastNode = ensureCapacity(1);
    // 设置节点位置信息对象到 triplets[3 * lastNode] 中
    setStorageInfo(lastNode, storage);
    // 设置下一个块为 null 到 triplets[3 * lastNode + 2]
    setNext(lastNode, null);
```



```

// 设置前一个块为 null 到 triplets[3 * lastNode + 1]
setPrevious(lastNode, null);
return true;
}

private void setStorageInfo(int index, DatanodeStorageInfo storage) {
    assert this.triplets != null : "BlockInfo is not initialized";
    assert index >= 0 && index*3 < triplets.length : "Index is out of bound";
    triplets[index*3] = storage;
}

private BlockInfoContiguous setPrevious(int index, BlockInfoContiguous to) {
    assert this.triplets != null : "BlockInfo is not initialized";
    assert index >= 0 && index*3+1 < triplets.length : "Index is out of bound";
    BlockInfoContiguous info = (BlockInfoContiguous)triplets[index*3+1];
    triplets[index*3+1] = to;
    return info;
}

```

另外一个操作是把此块的信息加入到当前维护的链表中，将 head 头节点 blocklist 以参数的形式传入，然后将返回值重新赋值给头节点，相当于进行了一次头节点的更新。

```

// 此处返回的 blockList 为新的头节点
blockList = b.listInsert(blockList, this);

```

listInsert 方法具体操作如下：

```

BlockInfoContiguous listInsert(BlockInfoContiguous head,
    DatanodeStorageInfo storage) {
    // 在当前块中寻找对应节点位置的下标
    int dnIndex = this.findStorageInfo(storage);
    assert dnIndex >= 0 : "Data node is not found: current";
    assert getPrevious(dnIndex) == null && getNext(dnIndex) == null :
        "Block is already in the list and cannot be inserted.";
    this.setPrevious(dnIndex, null);
    // 将当前的下一节点指向头节点
    this.setNext(dnIndex, head);
    if(head != null)
        // 将头节点的前一节点指向当前节点
        head.setPrevious(head.findStorageInfo(storage), this);
    // 返回当前节点为新的头节点
    return this;
}

```

在之前 addStorage 方法中设置的 null 会在此操作中连向 head 头节点。此部分过程见图 6-8。

2. listRemove

另外一个对应的操作是节点的 removeBlock 动作。在节点上执行了删除块动作之后，

会触发以下链表操作：

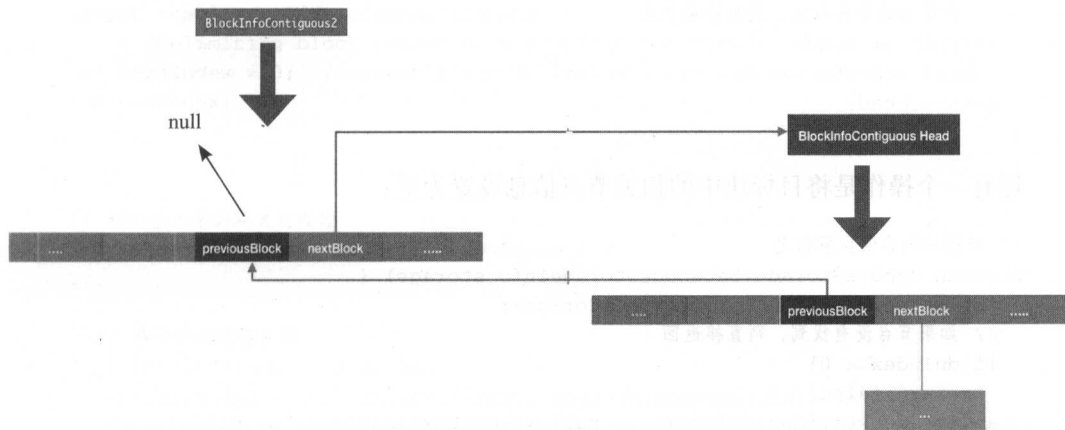


图 6-8 块添加操作流程

```
public boolean removeBlock(BlockInfoContiguous b) {
    blockList = b.listRemove(blockList, this);
    if (b.removeStorage(this)) {
        numBlocks--;
        return true;
    } else {
        return false;
    }
}
```

同样会有两个步骤，第一从链表中移除掉目标块，第二从目标块自身中释放掉关于其存储节点的信息。首先来看 `listRemove` 将当前目标块清除的操作：

```
BlockInfoContiguous listRemove(BlockInfoContiguous head,
    DatanodeStorageInfo storage) {
    if (head == null)
        return null;
    int dnIndex = this.findStorageInfo(storage);
    // 此块不存在于当前的节点列表中
    if (dnIndex < 0)
        return head;

    // 将对应的当前节点信息置为空
    BlockInfoContiguous next = this.getNext(dnIndex);
    BlockInfoContiguous prev = this.getPrevious(dnIndex);
    this.setNext(dnIndex, null);
    this.setPrevious(dnIndex, null);
    // 将前后节点关联
    if (prev != null)
        prev.setNext(prev.findStorageInfo(storage), next);
}
```

```

if(next != null)
    next.setPrevious(next.findStorageInfo(storage), prev);
// 如果当前为头节点, 则替换头节点
if(this == head)
    head = next;
return head;
}

```

还有一个操作是将目标块中的相关节点信息设置为空:

```

// 移除块的存储位置信息
boolean removeStorage(DatanodeStorageInfo storage) {
    int dnIndex = findStorageInfo(storage);
    // 如果节点没有找到, 则直接返回
    if(dnIndex < 0)
        return false;
    assert getPrevious(dnIndex) == null && getNext(dnIndex) == null :
        "Block is still in the list and must be removed first.";
    // 寻找最后一个不为空的节点
    int lastNode = numNodes()-1;
    // 用最后一个节点取代当前节点
    setStorageInfo(dnIndex, getStorageInfo(lastNode));
    setNext(dnIndex, getNext(lastNode));
    setPrevious(dnIndex, getPrevious(lastNode));
    // 置空前连接块信息以及 DataStorageInfo 信息
    setStorageInfo(lastNode, null);
    setNext(lastNode, null);
    setPrevious(lastNode, null);
    return true;
}

```

这里的动作是将最后一个节点的位置替换到当前要删除的位置, 并将原最后节点置为空。这是为了方便后面在 `ensureCapacity` 方法中动态扩充 `triplets` 数组的大小时, 无需重新创建对象数组。

3. moveBlockToHead

`moveBlockToHead` 操作也是 `BlockInfoContiguous` 类中经常会被调用的方法, 此方法会在 HDFS 块上报处理的过程方法 `reportDiff` 中被调用。

```

private void reportDiff(DatanodeStorageInfo storageInfo,
    BlockListAsLongs newReport,
    Collection<BlockInfoContiguous> toAdd,
    Collection<Block> toRemove,
    Collection<Block> toInvalidate,
    Collection<BlockToMarkCorrupt> toCorrupt,
    Collection<StatefulBlockInfo> toUC) {

```

```

// 新建一个 delimiter 分隔块, 用于分离汇报上来的块以及那些没有被汇报上来的块

```



```

BlockInfoContiguous delimiter = new BlockInfoContiguous(new Block(), (short) 1);
AddBlockResult result = storageInfo.addBlock(delimiter);
assert result == AddBlockResult.ADDED
    : "Delimiting block cannot be present in the node";
int headIndex = 0; //currently the delimiter is in the head of the list
int curIndex;

...

// 对汇报上来的块进行处理
for (BlockReportReplica iblk : newReport) {
    ...

    // 移动块到链表头部
    if (storedBlock != null &&
        (curIndex = storedBlock.findStorageInfo(storageInfo)) >= 0) {
        headIndex = storageInfo.moveBlockToHead(storedBlock, curIndex, headIndex);
    }
}
...

```

原理是通过将块移动到标记块的一侧，用以区分哪些块在本轮被汇报过，moveBlockToHead的作用是将块直接移到链表头部：

```

// 移动块到链表头部
public BlockInfoContiguous moveBlockToHead(BlockInfoContiguous head,
    DatanodeStorageInfo storage, int curIndex, int headIndex) {
    if (head == this) {
        return this;
    }
    // 将当前块的下一节点指向头节点
    BlockInfoContiguous next = this.setNext(curIndex, head);
    // 置空前一节点
    BlockInfoContiguous prev = this.setPrevious(curIndex, null);

    // 设置头节点的前一节点为空
    head.setPrevious(headIndex, this);
    // 将当前节点原来的前后节点相连
    prev.setNext(prev.findStorageInfo(storage), next);
    if (next != null) {
        next.setPrevious(next.findStorageInfo(storage), prev);
    }
    return this;
}

```

此部分流程见图 6-9。

在 BlockInfoContiguous 类中，还有一些其他辅助方法，这里主要分析其中的 3 种方法，也是经常被调用的 3 种方法。图 6-10 所示的是 BlockInfoContiguous 中所有的操作方法。

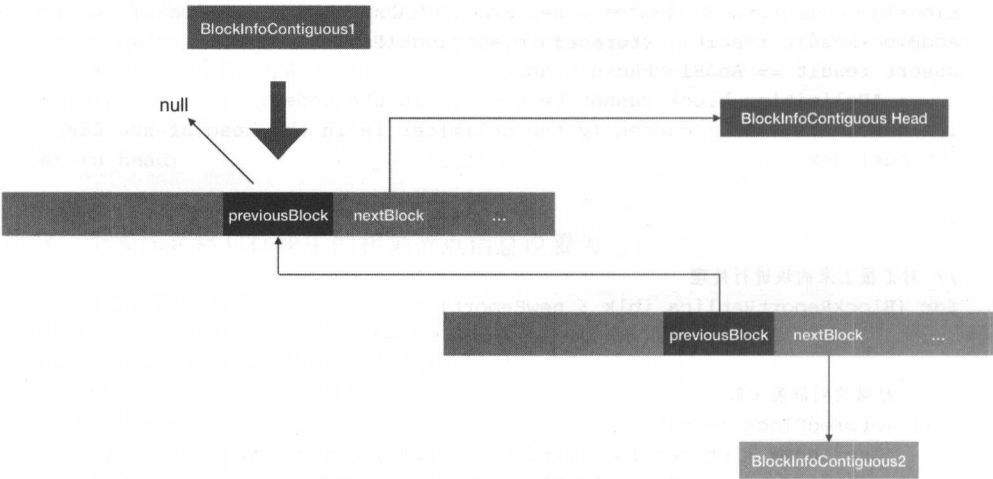


图 6-9 moveBlockToHead 流程图

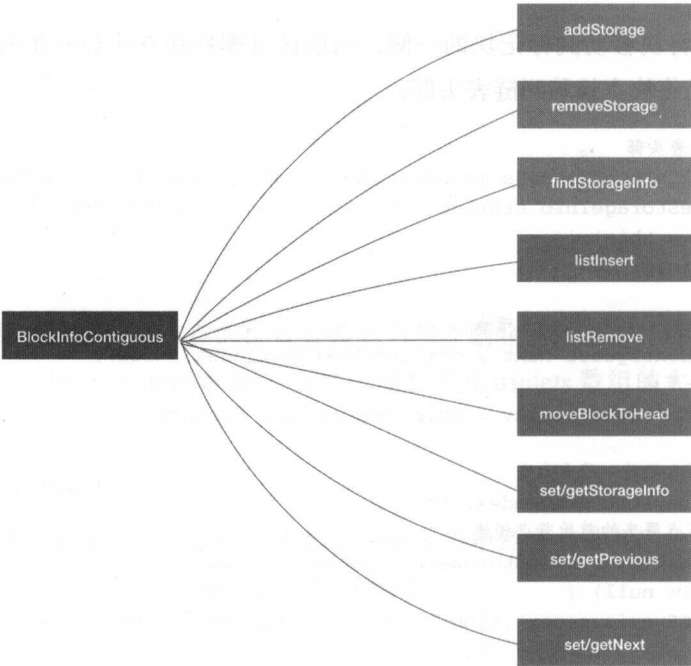


图 6-10 BlockInfoContiguous 内部的操作方法

6.3.3 块迭代器 BlockIterator

对于一个节点，我们想要遍历上面的块，就需要一个迭代器，能够通过类似 `next` 的方法获取其中的块。在 JDK 自带的链表中是有直接获取的方法的，但是对于 HDFS 为块设计

的这套特殊链表中，我们需要自己额外构造相应的迭代器。HDFS 的内部也的确设计了这样的迭代器，下面是总迭代器类，代码如下：

```
// DataNode 内部的总迭代器，包含了每个存储目录下的迭代器对象
private static class BlockIterator implements Iterator<BlockInfoContiguous> {
    private int index = 0;
    // DataNode 存储目录对应的迭代器列表
    private final List<Iterator<BlockInfoContiguous>> iterators;

    private BlockIterator(final DatanodeStorageInfo... storages) {
        List<Iterator<BlockInfoContiguous>> iterators = new ArrayList<Iterator<Block
            InfoContiguous>>();
        for (DatanodeStorageInfo e : storages) {
            iterators.add(e.getBlockIterator());
        }
        this.iterators = Collections.unmodifiableList(iterators);
    }

    // 判断是否还有下一个块
    public boolean hasNext() {
        update();
        return !iterators.isEmpty() && iterators.get(index).hasNext();
    }

    // 获取下一个块方法
    public BlockInfoContiguous next() {
        update();
        return iterators.get(index).next();
    }

    // 暂不支持移除方法
    public void remove() {
        throw new UnsupportedOperationException("Remove unsupported.");
    }

    // 更新跳过一个块
    private void update() {
        while(index < iterators.size() - 1 && !iterators.get(index).hasNext()) {
            index++;
        }
    }
}
```

Storages 节点信息是以参数的形式传入的：

```
DatanodeStorageInfo[] getStorageInfos() {
    synchronized (storageMap) {
        final Collection<DatanodeStorageInfo> storages = storageMap.values();
        return storages.toArray(new DatanodeStorageInfo[storages.size()]);
    }
}
```

具体迭代器的内部设计如下:

```
// DataNode 内部针对每个存储目录的块遍历迭代器
class BlockIterator implements Iterator<BlockInfoContiguous> {
    private BlockInfoContiguous current;

    BlockIterator(BlockInfoContiguous head) {
        this.current = head;
    }

    public boolean hasNext() {
        return current != null;
    }

    public BlockInfoContiguous next() {
        BlockInfoContiguous res = current;
        current = current.getNext(current.findStorageInfo(DatanodeStorageInfo.this));
        return res;
    }

    public void remove() {
        throw new UnsupportedOperationException("Sorry. can't remove.");
    }
}
```

在 DecommissionManager 的 processForDecomInternal 方法中就用到了这个迭代器:

```
private AbstractList<BlockInfoContiguous> handleInsufficientlyReplicated(
    final DatanodeDescriptor datanode) {
    AbstractList<BlockInfoContiguous> insufficient = new ChunkedArrayList<>();
    // 调用 DataNode 迭代器进行下线节点中块的遍历
    processBlocksForDecomInternal(datanode, datanode.getBlockIterator(),
        insufficient, false);
    return insufficient;
}
```

以上内容就是本节主要讲述的关于 HDFS 块链表方面的内容,也帮大家复习了数据结构中的常见链表操作。这里需要提醒一点,一旦集群中的块数达到千万级别,BlockInfoContiguous 对象同样会消耗掉大量的存储空间,也就是说同时会有千万个 INodeFile 和 BlockInfoContiguous 对象在 NameNode 的内存中。

6.4 小结

本章的内容并不算太多,大体上大家做到理解、会用即可,尤其像镜像文件相关的处理命令 hdfs oiv,有时候会帮我们不少忙。BlockInfoContiguous 类的设计与其对于块的组织是本章的一个难点,还需读者反复阅读、理解。

第三部分 *Part 3*

解决方案篇

HDFS 的数据管理

在本章中，我们将会学习到在真实应用环境中的一些实践经验。比如说对 HDFS 正常读写的限流方案，又比如说 HDFS 的数据规模的监控，以此帮助我们了解目前集群的一个数据统计情况。还有关于 HDFS 上的数据以及 DataNode 迁移的方案。最后是笔者在实践过程中做过的稍微简单一些的操作，比如 HDFS 的重命名方案以及配置化管理操作。

7.1 HDFS 的读写限流方案

HDFS 的读写限流指的是在 HDFS 普通读写文件块的操作中，对其速率进行限制，防止出现网络带宽打满的现象。在之前第 5.1 节中，我们讲述过 HDFS 内部的一些限流场景，但是对于普通的读写操作，HDFS 并没有做限流。本节所讲述的限流方案正是基于这个前提。本节将详细地介绍此次的限流方案，包括它的设计、实现以及最后的测试结果分析。

7.1.1 限流方案实现要点以及可能造成的影响

本节的使用场景为：集群大任务运行时，打满网络带宽，导致影响到其他业务方服务的运行。

以下列出的内容是限流方案实现中需要考虑的一些因素，以及这些因素在实现过程中可能对现有系统产生的影响。

对于限流方案，笔者总结出了以下实现要点：

□ 限流的操作对象应该是远程读和普通写操作，而不应该包括本地读的操作。在

HDFS 中，任何读写操作都会尽可能地选择本地的方式进行读写，这样可以避免网络数据传输，所以本地读在读操作中占的比例还是很高的。在这一点上，需要进行过滤，否则很容易“误伤”，导致无效的带宽限制。而普通的写操作我们大体上可以看作是分布式的写操作，直接限流就可以了。

- 需要新增动态调整限流带宽的管理命令。操作命令应该类似于设置带宽的命令 `dfsadmin -setBandwidth`。因为有的时候，我们对集群进行限流只是在特定的期间或特定的时段，其余正常的时段需要能够关闭限流功能。所以这里需要有动态的调整手段，而不是每次需要重启 `DataNode` 进程服务。
- IPC 通信的超时时间需要增大。这一点是之前在 HDFS 内部限流章节中没有提到的一点，这点也是笔者在真实测试使用时发现的问题。当限流功能打开的时候，因为一个写操作是以 Pipeline 的形式写到 3 个节点中的，所以限流操作会导致拖慢当前节点，进而拖慢此节点的下游节点。所以有的时候会引起 IPC 通信的超时，会出现 `Socket Timeout` 的现象。而且如果有很多个文件读写变慢，也会使集群整体的吞吐量下降。

7.1.2 限流方案实现

对于限流方案的实现，本节不会讲述具体代码实现的细节，在本节末尾，会给出一份 patch 文件的链接。这个 patch 的原型是 Hadoop 社区上的 JIRA：HDFS-9796 (Add throttler for datanode bandwidth)。基于这个最初的原型，我们加入了自身的额外需求 (patch 可能不能直接应用到别的版本的 HDFS 源码中，但是读者可以阅读 patch 代码做对照更改)。下面主要分析其中的细节原理以及实现逻辑：

- `DataXceiver` 的 `readBlock`、`writeBlock` 的限流。之前已经提到过，在 HDFS 中做普通读写的限流其实非常简单，只要在所有读写操作方法的前一步，传入一个限流器对象即可。这个限流器对象与 `Balancer` 数据限流器是同一个实现类。读操作经过的是 `readBlock` 方法，而写操作则是 `writeBlock`，所以我们要新建一个 `DataThrottler` 的限流器对象，以参数的形式传入这 2 个方法。同时在 `readBlock` 的时候，一定要判定是否为本地读的情况，如果是则传入 `null`，代表不限流。判断的核心是下面这行代码，通过 `Socket` 连接对象类型来做判断：

```
DataTransferThrottler dataThrottler =
    peer.isLocal() ? null : dataXceiverServer.dataThrottler;
```

- 新增动态设置限流带宽速率的命令。动态设置限流带宽命令的目的在上文中已经提到过了，还是非常有必要的，实现的原理几乎与 `dfsadmin` 带宽设置命令一致。通过 `dfsadmin` 带宽设置命令将目标带宽值发送给 `NameNode`，然后 `NameNode` 通过心跳

的方式将新的带宽值发送到各个 DataNode 上并进行更新。但是这种方式唯一的缺点是它是统一设置的命令，不会有差异性。在 Hadoop 中要想新增使用命令，需要添加新的 RPC 接口，而且需要更改 pb 协议。

7.1.3 限流测试结果

这个 patch 完成之后，笔者把它打入到了我们内部的 Hadoop 版本中，并重新进行了编译、打包、发布。随后我们发现了一些有意思的现象。

当我们只对其中的 1 台节点打开限流操作后，其影响很小，我们用 `hadoop fs -put` 大文件的方式进行测试，一切正常。而当我们对集群中所有的 hdfs 的 jar 包进行更新并全部限流到 10MB 时，试验结果还是很明显的，上传大文件的时间从之前的耗时几秒到测试时的几分钟。

后来我们在线上集群测试的结果能够显示出在中小规模集群中对 HDFS 进行限流还是能够起到一定作用的。但是注意这里有一个前提：中小规模集群。如果是集群节点数过万的大规模集群，可能会遇到如上文中提到的拖慢整个集群的问题。所以在这一点上，社区也没有将 HDFS-9796 合入到主干代码，原因也大致在于此。对于 HDFS 的内部限流，仍然需要更好的、考虑更周全的限流方案。最后给出笔者基于 HDFS-9796 改造后的代码 patch 链接：<https://github.com/linyiqun/open-source-patch/blob/master/hdfs/others/HDFS-dataThrottler/HDFS-dataThrottler.patch>。

Shuffle 限流

笔者使用上述方式在 HDFS 中进行限流，有的时候还是会出现个别时间点带宽飙升的场景，后来发现可能是 Reduce 任务 shuffle 数据导致的。因为 shuffle 的数据是 Map 任务产生的中间结果数据，中间结果数据是不走 HDFS 过程的，所以 HDFS 的限流对它其实不起作用。当然，如果你想对 shuffle 过程做限流，HDFS 这套限流逻辑也完全适用。但是这并不能确保完全限制流量，可能在未来某个时刻又会有未预料到的因素出现。所以最好的办法还是在 Hadoop 系统外部来做，比如对集群所在的机房做总带宽限制。

7.2 HDFS 数据资源使用量分析以及趋势预测

HDFS 资源使用量是集群监控统计中一项十分重要的指标数据。通过此数据，我们不仅可以了解集群每日新增的文件、目录数，还能知道集群每天写出了多少量的数据。通过不断采集资源使用量数据，我们可以预测出其中的变化趋势，然后做针对性地集群扩容等工

作。本节将为大家介绍获取 HDFS 资源使用量数据的方法，有了这些数据后，我们才能做后面的趋势预测。本节大致划分成 3 个部分来讲述：第一，要获取的指标数据；第二，获取数据的方法；第三，数据的使用。

7.2.1 要获取哪些数据

数据不是获取得越多、越细就越好，因为我们还要考虑其中的成本代价。在这里我们很明确，我们只需要一个宏观上的数据，总体使用情况的一些数据，比如 dfs 资源使用量等这类的数据信息。这些信息可以参考 NameNode 上的页面数据，图 7-1 所示即为可选的一些数据。

Summary	
Security is off.	
Safemode is off.	
714 files and directories,	2049 blocks = 12763 total filesystem object(s).
Heap Memory used 396.22 MB of 722 MB Heap Memory. Max Heap Memory is 869 MB.	
Non Heap Memory used 43.27 MB of 43.81 MB Committed Non Heap Memory. Max Non Heap Memory is 130 MB.	
Configured Capacity:	393.22 GB
DFS Used:	128.75 MB (0.03%)
Non DFS Used:	40.07 GB
DFS Remaining:	353.03 GB (89.78%)
Block Pool Used:	128.75 MB (0.03%)
DataNodes usages% (Min/Median/Max/stdDev):	0.01% / 0.01% / 0.10% / 0.04%
Live Nodes	4 (Decommissioned: 0)
Dead Nodes	1 (Decommissioned: 0)
Decommissioning Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0

图 7-1 NameNode 数据使用情况指标

这些数据足以用来展示目前集群的数据使用情况以及进行数据的未来分析。我们可以以这些数据为目标，想办法去获取这些数据。

7.2.2 如何获取这些数据

目标倒是有了，但是如何去获取这些数据呢？如果你想用 `hdfs` 命令行的方式去获取，答案当然是可以的，研究过 `hadoop-2.7.1` 源码的同学们应该知道，这个版本的发布包里包含了許多分析镜像文件的命令，这些命令可以离线计算出集群中文件块的信息，这些命令具体都是以 `hdfs oiv` 开头，`oiv` 就是 `OfflineImageView` 的缩写。但是这种方法有几个弊端：效率低，性能差。为什么这么说呢，因为分析镜像文件数据的时候，数据每次都需要重新计算，而且当镜像文件变得越来越大时，所需要的计算耗时也会随之增大。而且这个命令本身属于离线分析型的命令，不适合定时周期性地运行。假设我们想缩小数据获

取间隔, 30 分钟获取一下文件块数量, 怎么办? 所以最佳的办法还是以 NameNode 页面数据获取的方式去拿到这些数据。因为我们经常刷新页面就能得到更新过后的数据, 说明 NameNode 页面有自己的一套获取此方面数据的方法, 只要我们找到了这方面的代码, 基本就算大功告成了。这里我们省略如何找到这部分代码的过程, 直接给出结果。HDFS 的确没有在 dfsadmin 这样的命令中添加类似数据获取的命令, 也没有添加对应的 RPC 接口。笔者最后是通过控制此页面的 dfshealth.html 文件才找到这段代码的, 最终使用的是 http 请求来获取信息, 相关类代码如下:

```
// 此类用于获取集群页面上所展示的数据
@interfaceAudience.Private
class ClusterJspHelper {
    private static final Log LOG = LogFactory.getLog(ClusterJspHelper.class);
    public static final String OVERALL_STATUS = "overall-status";
    public static final String DEAD = "Dead";
    private static final String JMX_QRY =
        "/jmx?qry=Hadoop:service=NameNode,name=NameNodeInfo";

    // 获取集群健康报告
    ClusterStatus generateClusterHealthReport() {
        ...
    }

    // 获取下线节点报告信息
    DecommissionStatus generateDecommissioningReport() {
        ...
    }

    ...

    // 集群统计信息类
    static class ClusterStatus {
        /** Exception indicates failure to get cluster status */
        Exception error = null;

        // 集群状态统计信息
        // 集群 id
        String clusterid = "";
        // 集群总容量大小
        long total_sum = 0;
        // 集群可用空间大小
        long free_sum = 0;
        // 集群已使用空间大小
        long clusterDfsUsed = 0;
        // 集群非 dfs 数据使用的大小
        long nonDfsUsed_sum = 0;
        // 集群总文件目录总数
        long totalFilesAndDirectories = 0;
```

```

// 集群内所有 NameNode 信息
final List<NamenodeStatus> nnList = new ArrayList<NamenodeStatus>();

// NameNode 异常信息图
final Map<String, Exception> nnExceptions = new HashMap<String, Exception>();

...
}

// NameNode 统计信息类
static class NamenodeStatus {
    ...
}

...
}

```

当然不是所有的方法我们都需要，我们只需调用获取集群健康状况报告的方法：

```

// 集群健康报告生成
ClusterStatus generateClusterHealthReport() {
    // 初始化 ClusterStatus 对象
    ClusterStatus cs = new ClusterStatus();
    Configuration conf = new Configuration();
    List<ConfiguredNNAddress> nns = null;
    try {
        nns = DFSUtil.flattenAddressMap(
            DFSUtil.getNNServiceRpcAddresses(conf));
    } catch (Exception e) {
        // Could not build cluster status
        cs.setError(e);
        return cs;
    }

    // 遍历处理集群中的每个 NameNode
    for (ConfiguredNNAddress cnn : nns) {
        InetSocketAddress isa = cnn.getAddress();
        NamenodeMXBeanHelper nnHelper = null;
        try {
            nnHelper = new NamenodeMXBeanHelper(isa, conf);
            // 获取状态信息值
            String mbeanProps = queryMbean(nnHelper.httpAddress, conf);
            // 解析状态信息值到 NamenodeStatus 对象中
            NamenodeStatus nn = nnHelper.getNamenodeStatus(mbeanProps);
            if (cs.clusterid.isEmpty() || cs.clusterid.equals("")) { // Set clusterid
                only once
                cs.clusterid = nnHelper.getClusterId(mbeanProps);
            }
            cs.addNamenodeStatus(nn);
        } catch (Exception e) {
            // track exceptions encountered when connecting to namenodes

```

```

        cs.addException(isa.getHostName(), e);
        continue;
    }
}
return cs;
}

```

在 `NamenodeStatus` 这个类中就包含了我们想要的监控数据信息：

```

// 此类保存了集群内部基本的数据统计信息
static class NamenodeStatus {
    // 主机名
    String host = "";
    // NameNode 总容量
    long capacity = 0L;
    // 可用空间大小
    long free = 0L;
    // 目前空间使用量
    long bpUsed = 0L;
    // 非 HDFS 使用量
    long nonDfsUsed = 0L;
    // 文件目录总数
    long filesAndDirectories = 0L;
    // 块总数
    long blocksCount = 0L;
    // 丢失的块总数
    long missingBlocksCount = 0L;
    // 目前活跃的 DataNode 数量
    int liveDatanodeCount = 0;
    // 正在下线节点的数量
    int liveDecomCount = 0;
    // 已经处于 dead 状态的节点数
    int deadDatanodeCount = 0;
    // 已经下线完毕的节点数
    int deadDecomCount = 0;
    // http 地址
    URL httpAddress = null;
    // 软件版本信息
    String softwareVersion = "";
}

```

思路大致清楚，下面就是把这段程序搬出来，套入到自己的 java 程序中即可，样例程序会在本节末尾给出。

7.2.3 怎么用这些数据

现在来考虑一个实际的问题：有了这些数据我们怎么用。首先要明白一件事情，数据拿来是要存下来的。所以首先就是存入数据库中，间隔周期可以自己调。一般这样的数据比较倾向于用于分析数据增长走势，所以建议做成折线趋势图，可以分为两个维度：

1) 文件目录数、块数折线趋势图。因为这两个指标可以用于发现集群每日的写入量多少。一旦集群所持有的文件、块数太多,会导致 NameNode 内存吃紧。因为 NameNode 要维护如此庞大的元数据信息压力会很大。第二个指标块数的统计在一定程度上能反映出小文件的数量情况,因为一旦小文件多了,会对集群造成十分严重的影响。比如极端一点的情况:假设我们把很多不到 1MB 的文件写入集群,那这些不到 1MB 的文件必然是单独的 1 个块(一般我们不会把块大小设置成小于 1MB),然后会造成块增长趋势与文件、目录的数目增长趋势类似。即便存储空间并没有受到很大的影响,但是会对集群的性能造成影响。NameNode 因此会维护大量的块对象数据。而且 Hadoop 本身不适用于小文件的存储,因为每读一个文件它要启动一个新任务,需要一定的成本代价。一旦你的小文件过多,首先会启动大量的 Map 任务去读文件,1 个任务一般读 1 个文件。所以有些极端情况下,某个 Job 会有上千、上万个任务在运行。这个现象笔者在工作中也确实经历过。

2) dfs 磁盘空间使用量的趋势分析图。这个很好理解,我们就是想知道集群每天数据增长的总量是多少,方便未来在适当的时候增加机器来扩充集群的容量。

图 7-2、7-3 是针对以上两点的实际效果图,测试的数据来自于笔者的测试集群。

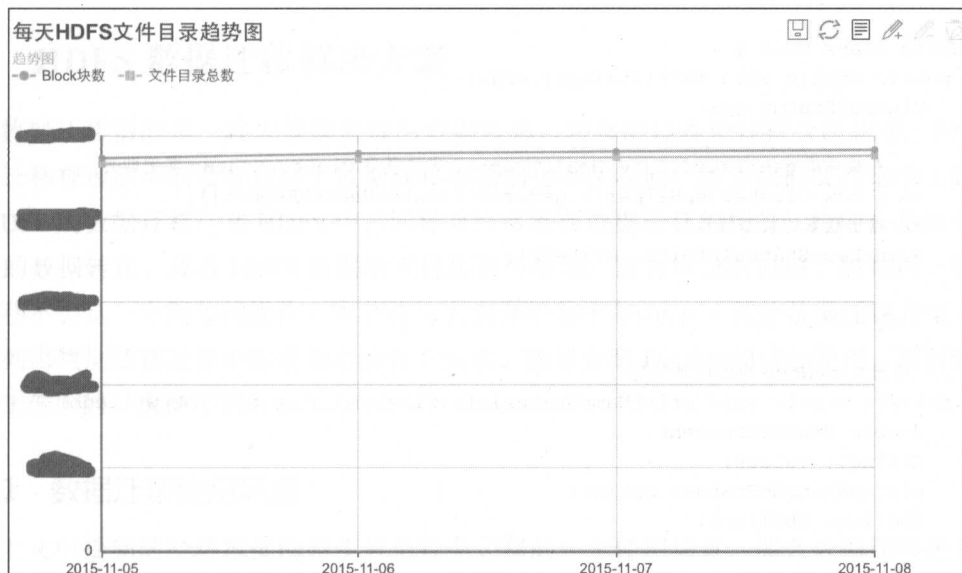


图 7-2 文件、块数量趋势图

这里有个小建议,在做预测分析图表的时候,最好把每天的统计数据 and 每小时的统计数据都计算一遍,一是能方便地知道集群内每天数据的增长量,二是我们能够了解到一天内哪些时段数据增长得比较快。

下面是获取统计数据的部分程序,在完整代码里还有获取 NodeManager 信息的程序:

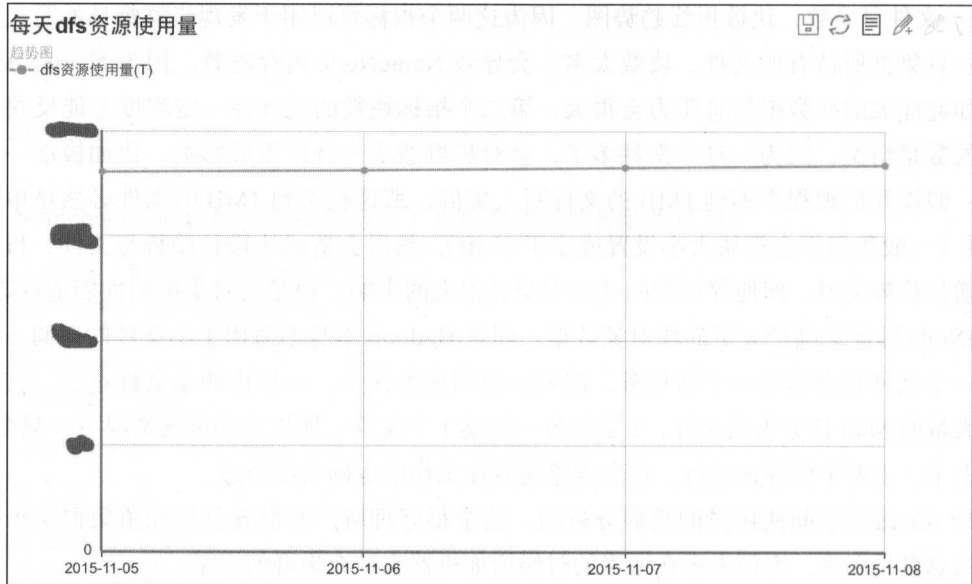


图 7-3 DFS 资源使用趋势图

```

public class Main {
    public static void main(String[] args) {
        ClusterStatus cs;
        ...
        // 直接调用 generateClusterHealthReport 方法获取 ClusterStatus 实例即可
        cs = new ClusterJspHelper().generateClusterHealthReport();
        // 输出获取的状态信息
        printNameStatusInfo(cs, writedb);
    }

    ...
    // 输出 NameNode 内的统计信息
    private static void printNameStatusInfo(ClusterStatus cs, int writedb) {
        double dfsUsedPercent;
        String[] values;
        List<NamenodeStatus> nsList;
        DbClient dbClient;
        ...

        nsList = cs.nnList;
        // 获取到 NameNode 的状态信息，并逐一输出
        for (NamenodeStatus ns : nsList) {
            values = new String[BaseValues.DB_COLUMN_CLUSTER_STATUS_LEN];
            dfsUsedPercent = 1.0 * ns.bpUsed / ns.capacity;
            // 输出信息内容含义可见上文中 NamenodeStatus 类中的注释信息
            System.out.println("host:" + ns.host);
        }
    }
}

```

```

System.out.println("blocksCount:" + ns.blocksCount);
System.out.println("bpUsed:" + ns.bpUsed);
System.out.println("capacity:" + ns.capacity);
System.out.println("deadDatanodeCount:" + ns.deadDatanodeCount);
System.out.println("deadDecomCount:" + ns.deadDecomCount);
System.out.println("filesAndDirectories:" + ns.filesAndDirectories);
System.out.println("free:" + ns.free);
System.out.println("liveDatanodeCount:" + ns.liveDatanodeCount);
System.out.println("liveDecomCount:" + ns.liveDecomCount);
System.out.println("missingBlocksCount:" + ns.missingBlocksCount);
System.out.println("nonDfsUsed:" + ns.nonDfsUsed);
System.out.println("dfsUsedPercent:" + dfsUsedPercent);
System.out.println("softwareVersion:" + ns.softwareVersion);

...
}
}

```

最后附上完整代码的链接:

<https://github.com/linyiquan/yarn-jobhistory-crawler/tree/master/NMTool>

7.3 HDFS 数据迁移解决方案

数据迁移指的是一种大规模量级的数据转移,转移的过程中往往会跨机房、跨集群。数据迁移规模的不同会导致整个数据迁移的周期也不尽相同。本节我们要讨论的主题是基于 HDFS 的数据迁移。在 HDFS 中,同样有许多需要数据迁移的场景,比如冷热数据集群之间的数据转化,或者 HDFS 数据的双机房备份等等。因为涉及跨机房、跨集群,所以数据迁移不会是一个简单的操作。本节将为大家介绍基于 DistCp 工具的数据迁移方案,首先将会列出数据迁移过程中需要考虑的各个因素,随后介绍 DistCp 的详细使用,最后给出笔者在工作中的一次操作实例。

7.3.1 数据迁移使用场景

上文中提到的冷热数据的同步只是数据迁移的一个使用场景,那么数据迁移还有其他哪些使用场景呢,这里总结出了如下几个:

- 冷热集群数据分类存储,详见上文描述。
- 集群数据整体搬迁。当公司的业务迅速发展,导致当前的服务器数量资源出现临时紧张的状况,为了更高效地利用资源,打算将原 A 机房数据整体迁移到 B 机房,原因是 B 机房机器多,而且 B 机房本身开销较 A 机房开销低些。
- 数据的准实时同步。数据的准实时同步与上一点的不同在于集群数据的整体搬迁可

以一次性操作解决，而准实时同步则需要定期同步，而且要做到周期内数据基本完全一致。数据准实时同步的目的在于数据的双备份可用，比如某天 A 集群突然宣告不允许再使用了，此时可以将线上的使用集群直接切向同步集群 B。因为 B 集群实时同步 A 集群数据，拥有几乎完全一致的真实数据和元数据信息，所以对于业务方的使用而言是不会受到任何影响的。

在上述 3 个使用场景中，第一个相比较于第二、三个来说可能稍微容易一些，但是想要完全做好也不简单。第三个数据的实时同步相比较第二个来说更加实际一些。因为如果公司准备做集群数据迁移了，一般都会提前通知，然后做逐步迁移，而且也肯定不会让原集群立即停止服务。一般采用数据慢慢同步的方式，等到数据彻底同步完毕，才最终实现切换，达到最终的迁移目标。

7.3.2 数据迁移要素考量

当准备做大规模数据迁移的时候，需要做很多前期准备工作，而且需要对很多因素、指标进行考量。以下是几个主要指标：

1. Bandwidth：带宽

在大规模数据的同步过程中，如何控制同步数据过程中所占用的网络带宽是十分重要的。带宽用得多了，会影响到线上业务的任务运行，带宽用的少了又会导致数据同步过慢。所以这里会引出另外一个问题：带宽的限流。也就是说，我们要保证数据同步程序在限定的网络传输速率下。如果我们不做任何处理的话，那结果基本上就是网络有多少带宽我们就用多少带宽的局面。

2. Performance：性能

性能问题同样也是一个很关键的问题，是采用简单的单机程序？还是性能更佳分布式程序？显然后者是我们更想要的。

3. Data-Increment：增量同步

当 TB、PB 级别的数据需要同步的时候，如果每次以全量的方式去同步数据，结果一定是非常糟糕的。增量方式的同步会是一个不错的选择，那么哪些情况下会导致数据发生增量变动呢？有如下两类情况：

- ❑ 原始数据文件进行了追加写。
- ❑ 原始数据文件被删除或重命名。

可能会有人好奇这里为什么没有对原始数据进行改动的情况，这种情况也会造成数据的变动。在海量数据存储系统中，例如 HDFS，一般不会对源文件内容上做修改，要么继续

追加写，要么删除文件，不会有类似 RandomAccessFile 的随机写的功能。所以做增量数据同步，只要考虑上述两个条件即可。上述条件中的第二点是非常容易判断的，通过定期的快照文件或元信息文件比较即可。但是对于文件是否进行了追加写或是其他外界主动修改操作的时候，我们如何进行判断呢，下面给出两个判断步骤：

1) 先比较文件大小，如果两个阶段文件大小发生改变，说明文件在内容上已经发生变更，变更的类型有两类。截取对应原始长度部分进行 checksum（校验和）比较，如果此 checksum 不变，则此文件发生了追加写的动作。如果 checksum 发生改变，说明文件在原内容上也已经改变。

2) 如果文件大小一致，则计算相应的 checksum。然后比较两者的 checksum，如果两者 checksum 一致，则说明文件内容没有发生改变。

这种方式的比较算得上是最保险的。

4. Syncable: 数据迁移的同步性

数据迁移过程中需要保证周期内数据一定能够同步完毕，不能差距太大。比如 A 集群 7 天内的增量数据，我们只要花半天就可以完全同步到 B 集群，然后又又可以等到下周再次进行同步。最可怕的事情在于 A 集群 7 天内的数据，我们的程序花了 7 天时间还同步不完，然后下一个周期又来了，这样就无法做到准实时的一致性。其实 7 天还是一个比较大的时间，最好是能达到按天同步。

7.3.3 HDFS 数据迁移解决方案：DistCp

上文分析了很多数据迁移的使用场景和可能出现的问题。但是从这里开始，将主要讲述 HDFS 中的数据迁移解决方案。面对上文中提到的诸多问题，在 HDFS 中到底应该如何解决呢？如果你不是 HDFS、Hadoop 的专家，可能问题看起来有点棘手，但是没有关系，Hadoop 内部专门开发了相应的工具：DistCp。DistCp 工具在 Hadoop 中的定位就是用于数据迁移的，针对的就是从源文件系统到目标文件系统的数据拷贝。DistCp 在 hadoop-tools 工程下，作为独立子工程存在。在官方注释中，对于 DistCp 的解释如下：

在命令行的操作使用上，DistCp 的 main 方法会解析命令行中输入的参数，然后会启动一个 DistCp 任务。而在程序层面的使用上，一个 DistCp 对象可以通过指定 DistCpOptions 内定义的一些参数进行构造，然后调用 DistCp 的 execute 方法来启动一个拷贝任务。

大意是通过命令行附带参数的形式，构造出 DistCp 的 Job，然后执行此 Job。从这里可以知道，拷贝任务本身是一个 MR 的 Job，已经把 Hadoop 本身的分布式执行的特性用上了。

7.3.4 DistCp 优势特性

鉴于 DistCp 的特殊使用场景，程序设计者在此工具代码中添加了很多独到的设计。下面针对上文提到的一些实现要点进行相应的讲述。

1. 带宽限流

DistCp 是支持带宽限流的，使用者可以通过命令参数 `bandwidth` 来为程序进行限流，原理类似于 HDFS 中数据 Balancer 程序的限流。但是笔者认为 DistCp 做得比 Balancer 稍微简化了一些。DistCp 中的相关类是 `ThrottledInputStream`，在每次读操作的时候，做一次限流判断：

```
/** {@inheritDoc} */
@Override
public int read() throws IOException {
    // 此处做一次限流判断
    throttle();
    int data = rawStream.read();
    if (data != -1) {
        bytesRead++;
    }
    return data;
}
```

然后在 `throttle` 的方法中进行当前传输速率的判断，如果速率过快会进行一段时间的睡眠来降低总平均速率：

```
private void throttle() throws IOException {
    // 如果计算出当前的平均速率已经超过最大传输字节速率，则将会睡眠一段时间
    while (getBytesPerSec() > maxBytesPerSec) {
        try {
            Thread.sleep(SLEEP_DURATION_MS);
            totalSleepTime += SLEEP_DURATION_MS;
        } catch (InterruptedException e) {
            throw new IOException("Thread aborted", e);
        }
    }
}
```

Balancer 内部的限流原理，可以查阅之前第 5.1 节的内容。

2. 增量数据同步

对于增量数据同步的需求，在 DistCp 中也得到了很好地实现。通过 `update`、`append` 和 `diff` 这 3 个参数能很好地解决。官方的参数使用说明如下：

□ **Update**：更新目标路径，只拷贝相对于源端，目标端不存在的文件或者目录。

□ **Append**：追加写目标路径下已存在的文件，如果这个文件在源端已经发生了追加写

操作。

□ Diff: 通过快照的 diff 对比信息来同步源端路径与目标路径。

第一个参数, 解决了新增文件、目录的同步。第二个参数, 解决已存在文件的增量更新同步。第三个参数解决删除或重命名类型文件的同步。这里 diff 参数的使用需要设置 2 个不同时间点的快照进行对比, 产生相应的 DiffInfo 信息。在获取快照文件的变化时, 只会选择 DELETE 和 RENAME 这两种类型的变化信息。

```
static DiffInfo[] getDiffs(SnapshotDiffReport report, Path targetDir) {
    List<DiffInfo> diffs = new ArrayList<>();
    for (SnapshotDiffReport.DiffReportEntry entry : report.getDiffList()) {
        // 只判断删除和重命名的类型
        if (entry.getType() == SnapshotDiffReport.DiffType.DELETE) {
            final Path source = new Path(targetDir,
                DFSUtil.bytes2String(entry.getSourcePath()));
            diffs.add(new DiffInfo(source, null));
        } else if (entry.getType() == SnapshotDiffReport.DiffType.RENAME) {
            final Path source = new Path(targetDir,
                DFSUtil.bytes2String(entry.getSourcePath()));
            final Path target = new Path(targetDir,
                DFSUtil.bytes2String(entry.getTargetPath()));
            diffs.add(new DiffInfo(source, target));
        }
    }
    return diffs.toArray(new DiffInfo[diffs.size()]);
}
```

在文件数据追加写的判断逻辑上, DistCp 程序做了很精细的判断。当文件大小不变的情况时, 首先判断是否可以跳过当前文件:

```
private boolean canSkip(FileSystem sourceFS, FileStatus source,
    FileStatus target) throws IOException {
    if (!syncFolders) {
        return true;
    }
    boolean sameLength = target.getLen() == source.getLen();
    boolean sameBlockSize = source.getBlockSize() == target.getBlockSize()
        || !preserve.contains(FileAttribute.BLOCKSIZE);
    // 如果是同大小并且 blockSize 的大小也一样, 则继续进行 checksum 的判断
    if (sameLength && sameBlockSize) {
        return skipCrc ||
            DistCpUtils.checksumsAreEqual(sourceFS, source.getPath(), null,
                targetFS, target.getPath());
    } else {
        return false;
    }
}
```

其次是判断是否可以进行追加写:

```

// 判断是否可以跳过此文件
if (canSkip(sourceFS, source, targetFileStatus)) {
    return FileAction.SKIP;
} else if (append) {
    // 如果是设置了追加写的方式, 首先获取源目标文件的大小
    long targetLen = targetFileStatus.getLen();
    // 如果源目标文件大小小于现在的源文件大小, 说明源文件进行了新的写操作
    if (targetLen < source.getLen()) {
        // 计算源文件中对应目标文件大小的文件校验和
        FileChecksum sourceChecksum = sourceFS.getFileChecksum(
            source.getPath(), targetLen);
        // 如果源文件对应长度的数据的校验和与目标文件校验和完全一致,
        // 表明源文件多出的数据完全是新写入的, 前面的数据没有变动, 支持追加写
        if (sourceChecksum != null
            && sourceChecksum.equals(targetFS.getFileChecksum(target))) {
            // We require that the checksum is not null. Thus currently only
            // DistributedFileSystem is supported
            return FileAction.APPEND;
        }
        // 如果校验和发生了变化, 说明源文件前面部分的数据发生了变动, 则将会进行
        // OVERWRITE 覆盖的动作
    }
}
}
return FileAction.OVERWRITE;
...

```

此处程序并没有完全将文件大小的变化作为根本依据, 大小发生了变化了, 还要再对之前对应长度的数据做校验和的验证。

在使用 DistCp 相关参数做增量数据同步的时候, 需要留意部分参数之间的互斥性。比如 -diff 参数是根据快照数据做同步的, 会同步 RENAME 和 DELETE 两类关系的变动。其中的 DELETE 操作与 -delete 参数有重复的作用, 所以建议 -diff 参数应该与 -update 参数结合使用, 而 -delete 参数则可以单独使用。

3. 高效的性能

关于 DistCp 的性能问题笔者想主要分析一下。因为带宽限流和增量的数据同步通过普通的程序优化也能够实现。但是在性能特性上, 笔者认为 DistCp 还是有它独到的优势的。

(1) 执行的分布式特性

之前在上文中已经提到过, DistCp 本身会构造成一个 MR 的 Job。它是一个纯由 Map 任务构成的 Job, 注意它是没有 Reduce 过程的。所以它能够把集群资源利用起来, 集群闲下来的资源越多, 它运行得就越快。下面是 DistCp Job 的构造过程:

```

// 根据当前配置创建一个新的 Job
private Job createJob() throws IOException {

```

```

// DistCp Job统一名称为“distcp”
String jobName = "distcp";
String userChosenName = getConf().get(JobContext.JOB_NAME);
if (userChosenName != null)
    jobName += ":" + userChosenName;
Job job = Job.getInstance(getConf());
job.setJobName(jobName);
job.setInputFormatClass(DistCpUtils.getStrategy(getConf(), inputOptions));
job.setJarByClass(CopyMapper.class);
configureOutputFormat(job);
// 设置特殊定制的 CopyMapper 的 map 类型
job.setMapperClass(CopyMapper.class);
// 无 Reduce Task
job.setNumReduceTasks(0);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
job.setOutputFormatClass(CopyOutputFormat.class);
job.getConfiguration().set(JobContext.MAP_SPECULATIVE, "false");
job.getConfiguration().set(JobContext.NUM_MAPS,
    String.valueOf(inputOptions.getMaxMaps()));

if (inputOptions.getSslConfigurationFile() != null) {
    setupSSLConfig(job);
}

inputOptions.appendToConf(job.getConfiguration());
// 返回构造好的 Job
return job;
}

```

(2) 高效的 MR 组件

高效的 MR 组件的指的是 DistCp 在相应的 Job 中提供了针对此类型任务独有的 Map 类、InputFormat 和 OutputFormat，分别是 CopyMapper、DynamicInputFormat 和 CopyOutputFormat。这三种 MR 组件类型与普通的 MR 类型有什么区别呢？答案如下所示：

- 在 HDFS 上拆分拷贝列表到更小粒度单元的 chunk。
- 创建一个空的动态的 spilt 分片，以此让每个任务可以消费尽可能多的 chunk。

以上强调了两点，DynamicInputFormat 类会将输入文件分成很多小的 chunk，然后由这些 chunk 构成动态的分片，尽可能地让 Map 任务消费掉。而不是按照传统的方式将输入文件分割成固定的分片。前者不会造成任何慢的 Map 任务拖累整个 Job 的运行。保证了哪个 Map 任务消费得快，那就消费更多分片的原则。其中具体的原理实现读者可自行到 org.apache.hadoop.tools.mapred.lib 包下的代码中进行分析。图 7-4 为 DistCp Job 的组成结构。

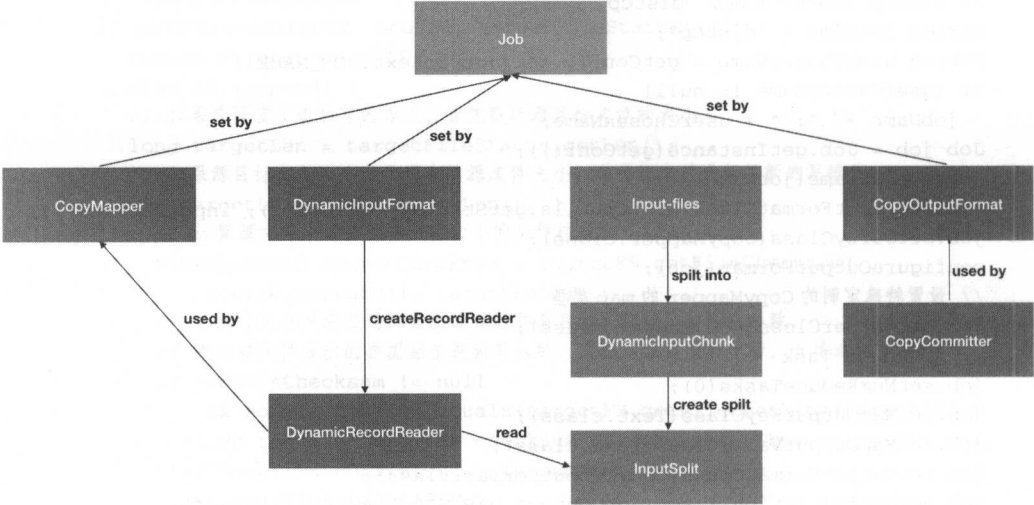


图 7-4 DistCp Job 结构图

7.3.5 Hadoop DistCp 命令

前面花了大量的篇幅讲述了 DistCp 工具的强大用处，最后给出使用帮助信息，输入 `hadoop distcp` 命令即可获得，以下列出几个常用的操作命令：

```
$ hadoop distcp
usage: distcp OPTIONS [source_path...] <target_path>
      OPTIONS
      -append           // 拷贝文件时支持对现有文件进行追加写操作
      -async            // 异步执行 distcp 拷贝任务
      -bandwidth <arg> // 对每个 Map 任务的带宽限速
      -delete           // 删除相对于源端，目标端多出来的文件
      -diff <arg>      // 通过快照 diff 信息进行数据的同步
      -overwrite        // 以覆盖的方式进行拷贝，如果目标端文件已经存在，则直接覆盖
      -p <arg>          // 拷贝数据时，扩展属性信息的保留，包括权限信息、块大小信息等等
      -skipcrccheck      // 拷贝数据时是否跳过校验和的校验
      -update           // 拷贝数据时，只拷贝相对于源端，目标端不存在的文件数据
```

其中 `source_path`、`target_path` 需要带上地址前缀以区分不同的集群，例如：

```
hadoop distcp hdfs://nn1:8020/foo/a hdfs://nn2:8020/bar/foo
```

上面的命令表示从 `nn1` 集群拷贝 `/foo/a` 路径下的数据到 `nn2` 集群的 `/bar/foo` 路径下。总体而言，DistCp 的可选参数还是做到了相当细粒度的控制，比如 `skipcrccheck` 的选项，可以跳过校验和的校验。跳过校验和可能会影响到 DistCp 数据完整性的判断，但同时此配置的关闭会使得拷贝过程更加高效。

当然跨机房数据迁移的工作一定还会出现没有预见到的问题，其中的难度和困难绝对

是非常具有挑战性的。可能我们还要利用 DistCp 的功能然后搭配上自己的解决方案才能做出更棒的方案。

7.3.6 DistCp 解决集群间数据迁移实例

下面分享的是笔者在实际工作中用 distcp 命令做不同数据中心之间数据同步的实际例子。数据同步的场景分为以下两类：

第一类，源端集群数据不会发生变更，是一堆静态数据。这种同步方式很简单，用 distcp 命令指明好源集群、目标集群地址，对想要同步数据的最顶层目录进行同步即可。

第二类，源端集群数据会时时变化更新。这种情况下，可以采用快照的方式进行同步：

- 1) 首先在源集群的目标拷贝目录上创建一个快照。
- 2) 然后利用 distcp 同步此快照到目录集群，第一轮同步过程是最花时间的，所以需要等待一段时间。
- 3) 等第一轮的快照同步结束之后，可以在源集群上的同样目录下创建第二个快照，再次做数据同步，但是这个时候可以使用一些 -delete、-update 的参数来同步发生变更的数据文件，而不是所有文件再次进行扫描同步。

4) 以此类推，当最终源端创建的快照与目标集群的数据几乎没有太大差别的时候，就可以做一次切换，将集群地址从源集群切到目标集群。

笔者在做数据同步的过程中也不是一帆风顺的，其中也踩了不少坑。

比如说在拷贝数据文件的时候，如果没有仔细注意拷贝路径的设置，会很容易造成拷贝文件位置出错的情况，比如文件拷贝到了目标路径的子目录中。第二类情况是集群中丢失的块会导致 distcp 拷贝任务的失败退出。这个现象非常隐蔽，丢失的块在一般情况下只会导致部分任务的数据拷贝失败，但是 distcp 程序会把在运行的任务全部停止运行。在这点上，笔者觉得此做法稍显不妥，毕竟 distcp 启动运行一次成本是比较高的。所以在数据同步的过程中，要及时留意 NameNode 页面上是否有丢失或损坏的块。如果存在则尽快用 fsck <path> -delete 进行删除。在数据同步的过程中，还需要注意数据权限等属性信息的保留，可以用 -p 加上想要保留属性的对应参数。因为如果没有保留属性信息的话，拷贝到目标集群的数据在运行任务的时候可能会造成文件所属用户读写数据失败的情况。

7.4 DataNode 迁移方案

本节标题所示的 DataNode 迁移并不是指 DataNode 上的数据迁移，而是指 DataNode 自身节点的搬迁。在搬迁的过程中，此节点将会停止服务，搬迁完成后，还可能涉及主机名、ip 地址的变化。本节所要讲述的就是此过程的迁移方案。一种情况是更换主机名、ip 地址

的情况，另一种是不更新主机名、ip 地址情况。本节介绍第一种情况。

7.4.1 迁移方案的目标

1. 目标

由于外界因素的影响，需要将原 DataNode 所在节点的机器从 A 机房换到 B 机房，其中会涉及主机名和 ip 地址的改变。最终的目标是 DataNode 迁移之后对集群不造成大的影响，服务依然可用，数据不发生丢失。

2. 相关知识

因为在 DataNode 迁移的时候，必定会导致迁移节点停止心跳，如果超过心跳检查超时时间，此节点就会被 NameNode 认为是死节点。为了满足块副本数要求，会造成集群内大量块复制的现象。如果想要在短时间内不使节点成为死节点，需要人工把心跳超时检查时间设大。NameNode 超时心跳检测时间算法如下：

```
DatanodeManager(final BlockManager blockManager, final Namesystem namesystem,
    final Configuration conf) throws IOException {
```

```
.....
```

```
    final long heartbeatIntervalSeconds = conf.getLong(
        DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
        DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT);
    final int heartbeatRecheckInterval = conf.getInt(
        DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_KEY,
        DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INTERVAL_DEFAULT); // 5 minutes
    // 心跳检测超时时间的计算
    this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval
        + 10 * 1000 * heartbeatIntervalSeconds;
```

```
.....
```

核心公式如下：

```
this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval +
10 * 1000 * heartbeatIntervalSeconds
```

heartbeatRecheckInterval 心跳检测时间默认 300 秒，心跳间隔时间 3 秒，默认超时时间 $2 \times 300 + 10 \times 3 = 630$ 秒，因此需要将前者配置加大。

```
</property>
<name>dfs.namenode.heartbeat.recheck-interval</name>
<value>10800000</value>
</property>
```

上述的配置表明将此值调大为 3 小时，这个值可以根据使用场景进行变化。执行以下操作可以使此配置值生效。

❑ 更新 standby namenode 的 hdfs-site.xml 的配置，并重启。

❑ 等待 standby namenode 退出 safemode 之后，再 stop active namenode，更新配置并重启。

但是此种方案适用于 DataNode 不涉及主机名和 ip 地址变化的情况。

下面是涉及主机名、ip 地址变化情况的迁移方案。

7.4.2 DataNode 更换主机名、ip 地址时的迁移方案

步骤 1: DataNode 迁移测试

在 DataNode 做迁移之前，进行测试文件的上传和 RPC 请求的测试，为了与后面的测试结果进行对比。

首先上传 1 个 test 文件：

```
bin/hadoop fs -put test.txt /tmp
```

保证此文件所在的块存在于此节点上，用 -cat 命令进行查看：

```
bin/hadoop fs -cat /tmp/test.txt
```

测试完毕，此时停止 DataNode，并使用 jps 命令查看此 DataNode 是否真正停止。

并观察 NameNode 的 Web 界面，在时间超过 630 秒后，迁移节点将被显示为 dead 状态，意为死节点。

之后在 NameNode Web 界面的待复制块（Number of Under-Replicated Blocks）指标将会显示正在进行拷贝的块副本数，表明目前有大量的块在进行副本复制。

步骤 2: DataNode 重启

重新启动 DataNode，查看 DataNode 输出日志。DataNode 在初次启动的时候由于缓存的 dfsUsed 值超过 600 秒会过期，需要重新执行 du 命令扫描 DataNode 上面的磁盘块进行 dfsUsed 使用量的计算，会消耗几分钟的时间（如果是立即停止，并马上重启 DataNode，将会非常快），日志如下：

```
2016-01-06 16:05:08,181 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Scanning block pool BP-1942012336-xx.xx.xx.xx-1406726500544
on volume /home/data/data/data/hadoop/dfs/data/data11/current...
2016-01-06 16:05:08,181 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Scanning block pool BP-1942012336-xx.xx.xx.xx-1406726500544
on volume /home/data/data/data/hadoop/dfs/data/data12/current...
2016-01-06 16:09:49,646 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Time taken to scan block pool BP-1942012336-xx.xx.xx.xx-
1406726500544 on /home/data/data/data/hadoop/dfs/data/data7/current: 281466ms
2016-01-06 16:09:54,235 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Time taken to scan block pool BP-1942012336-xx.xx.xx.xx-
1406726500544 on /home/data/data/data/hadoop/dfs/data/data9/current: 286054ms
```

出现上述“Time taken”开头的日志代表磁盘扫描操作结束，DataNode 启动成功了。

❑ DataNode 成功启动后，NameNode 页面上的待复制块指标将会重新变为 0 或比较小的一些数值，比如几十或几百，分别代表不需要和极少需要额外副本块的复制，这些都属于正常的情况。

❑ 在此节点上重新执行 `bin/hadoop fs -cat /tmp/test.txt` 命令，测试文件内容是否还能够查看，测试结束后删除测试文件。

迁移方案总结：

❑ 对于更换主机名和 ip 地址的 DataNode 迁移操作而言，只要在可控的时间内恢复 DataNode 服务即可，不会对集群造成大的影响，只会在迁移节点成为死节点的状态时才会出现短暂块复制的现象。

❑ 对于不变化主机名和 ip 地址情况的 DataNode 迁移操作，只要加大心跳检测时间，使其在短时间内不成为死节点，然后进行恢复即可，此操作将不会对集群造成影响。心跳检测时间配置项为 `dfs.namenode.heartbeat.recheck-interval`，默认单位为毫秒。

7.5 HDFS 集群重命名方案

在部分集群数据迁移之后，有时需要维持新老集群名称的一致，否则会出现客户端程序读写数据异常的现象。但是由于在搭建新集群的时候并未预见此状况的发生，已经确定好了集群名称。这个时候我们该怎么办？本节所要讲述的 HDFS 集群重命名方案就是为了解决这个问题。本节通过操作步骤的形式来讲述重命名方案，在每个步骤中会介绍具体的操作行为或执行命令。

第一步：停止集群所有服务

因为涉及 HDFS 集群名称的变更，所以 HDFS 相关的服务一定得停止，其次为了避免其他的影响，最好把 YARN 相关的服务也停止。

```
$sbin/stop-dfs.sh
$sbin/stop-yarn.sh
```

第二步：修改 HDFS 集群名称相关配置

此步骤中涉及了 3 个配置文件的变更，分别为 `core-site.xml`、`hdfs-site.xml` 和 `yarn-site.xml`。原集群名称假设为 `clusterA`，目标名称为 `clusterB`。下面为这 3 个文件的变更修改。

1) `core-site.xml`

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://clusterA</value>
```

```
<final>true</final>
</property>
```

此处将 `hdfs://clusterA` 替换为 `hdfs://clusterB` 即可。

2) yarn-site.xml

因为 YARN 部分应用相关的数据也会存放在 HDFS 之上，所以这部分的配置也是需要更新的。如下配置所示：

```
<property>
    <name>yarn.resourcemanager.fs.state-store.uri</name>
    <value>hdfs://clusterA/logs/yarn/rmstore</value>
</property>
```

同理，将 `clusterA` 进行替换。

3) hdfs-site.xml

`hdfs-site` 文件无疑是配置属性变更最多的一个文件。

```
<property>
    <name>dfs.nameservices</name>
    <value>clusterA</value>
</property>
<property>
    <name>dfs.ha.namenodes.clusterA</name>
    <value>nn1,nn2</value>
</property>
<property>
    <name>dfs.namenode.rpc-address.clusterA.nn1</name>
    <value>clusternn1:9000</value>
</property>
<property>
    <name>dfs.namenode.rpc-address.clusterA.nn2</name>
    <value>clusternn2:9000</value>
</property>
<property>
    <name>dfs.namenode.http-address.clusterA.nn1</name>
    <value>clusternn1:50070</value>
</property>
<property>
    <name>dfs.namenode.http-address.clusterA.nn2</name>
    <value>clusternn2:50070</value>
</property>
<property>
    <name>dfs.client.failover.proxy.provider.clusterA</name>
    <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxy
        Provider</value>
</property>
```

这里的 `clusterA` 全部替换为目标名称 `clusterB`。

自此，所有相关配置项都已更改完毕。但是这里还需要执行下一个关键的步骤。

第三步：重新格式化 HDFS 所依赖的 znode

这步操作是操作者经常容易忘记的，没有操作过的人可能不会想到，NameNode 会把自身的集群名称与 znode 进行关联。

如果没有重新执行 format znode 的操作，在重启 NameNode 的时候会出现下面的错误：

```
/10.11.2.81:2181, sessionId = 0x51535b491d5e0ab8, negotiated timeout = 30000
2016-05-16 15:27:21,087 INFO [main-EventThread] (ActiveStandbyElector.java:547)
- Session connected.
2016-05-16 15:27:21,095 FATAL [main] (ZKFailoverController.java:219) - Unable to
start failover controller. Parent znode does not exist.
Run with -formatZK flag to initialize ZooKeeper.
2016-05-16 15:27:21,103 INFO [main] (ZooKeeper.java:684) - Session:
0x51535b491d5e0ab8 closed
2016-05-16 15:27:21,103 INFO [main-EventThread] (ClientCnxn.java:512) -
EventThread shut down
```

其中提示 parent znode 没有找到，同时建议我们执行 -formatZK 的操作。这个错误会导致 zkfc 进程启动失败，继而两个 NameNode 的 Active、Standby 切换将会失败，你会看到同时出现两个 Standby NameNode 的场景。最后只能通过手动切换的方式选出一个节点作为 Active NameNode。为什么会出现这样的错误呢？这个得要亲自到 zk 上去查看，才能明白其中的原因，我们用 zk 客户端命令查看相关 znode 节点，结果如下：

```
[zk: xx.xx.xx.xx:2181(CONNECTED) 2] ls /hadoopTestCluster-hadoop-ha
[clusterA]
```

从这里可以看到，在 HDFS 所关联的 znode 子节点中，会把当前集群名称作为子节点的名称。再次查看里面存放的内容：

```
[zk: xx.xx.xx.xx:2181(CONNECTED) 8] get /hadoopTestCluster-hadoop-ha/clusterA/
ActiveBreadCrumb
clusterAxxxx F(>
cZxid = 0x90000000a3
ctime = Wed Mar 02 15:41:23 CST 2016
mZxid = 0x9002937dd
mtime = Mon May 16 15:40:02 CST 2016
pZxid = 0x90000000a3
cversion = 0
dataVersion = 17
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 33
numChildren = 0
```

在以集群名为名称的子节点中就存放了 Active NameNode 的主机名以及其他相关的信息。这个节点名称在第一次 zk format 的时候就确定下来了，倘若集群更换了名称，自然就

找不到对应的 `znode` 了。

所以要解决这个问题，还是要重新执行 `format znode` 操作。首先停止 HDFS 相关服务，然后执行 `format` 命令：

```
hdfs zkfc -formatZK
```

确认操作没有出现异常后，执行下面的验证操作。如果都进行顺利的话，HDFS 重命名操作就算成功了。

- 重启 HDFS 相关服务。
- 执行 `hadoop` 的 `fs` 命令是否可用。
- 最后启动 YARN 相关的服务，并提交一个 `wordcount` 任务到 YARN 上做测试。

7.6 HDFS 的配置管理方案

HDFS 配置管理方案所要解决的问题是 HDFS 的配置管理问题。在集群规模越发庞大的情况下，HDFS 的配置管理往往越难。在其间会遇到两大问题：第一，集群的异构性造成配置的多样化；第二，多人维护配置导致配置变更记录难以维护。本节将为大家介绍一种基于 Git 的简单配置管理方案。在此方案中，我们巧妙地利用了 Git 本身具有的历史版本记录的功能。在方案讲述的过程中，还将会介绍现有配置管理存在的问题以及目前用于配置管理的一些开源工具。

7.6.1 HDFS 配置管理的问题

当集群规模日益扩张，渐渐地我们会碰到一些配置管理上的问题。有人可能会有疑问，集群扩容的时候，都是用相同的安装包部署，配置不都是一样的吗？为什么还会碰到配置管理的问题？当看完这段文字，作为一名运维工程师，如果你也是这么想的话，那只能说明一个问题：你管理的集群规模还不够大或者是你的集群变更操作太少，节点部署完了，基本就不动它了（这里不包括常用的启动和停止服务操作）。下面来重点谈谈集群规模变大会给配置管理带来的两个主要问题。

- 配置的差异化管理。随着集群规模的扩大，会不断有新节点加入。但是新扩容节点的机型并不保证都与现有节点完全一致。而且有时我们也需要一些异构的机器来做相应的异构存储，比如最常见的冷热数据的分类存储。这时我们需要一些拥有高密度存储特性但计算性能不是很高的机器。机型的不同会带来一个直接的变化，就是配置的不同。以异构存储为例，如果集群中一部分机器被用做冷数据的存储，它的数据目录的配置一般会带上 `[ARCHIVE]` 标签。而普通机器在此配置项上就可以直接写存储目录的位置，这就会带来一个配置的差异。当然，还有更常见的内存、磁盘

数、核数等差异。而配置的差异化会给集群的管理带来不小的麻烦。因为此时集群运维人员需要维护多份配置，然后根据对应机型的节点来分发对应的配置，这样不利于做到批量处理。在多份配置的情况下，有的时候还会导致人工的误操作。

- 配置的历史记录管理。配置的历史记录管理指的是配置的变更管理。如果集群维护人员就只有 1、2 个人的时候，配置的变更可能不会出现很大的问题。但是如果集群是在多人维护的情况下，可能会发生配置变更信息不同步的问题。比如集群管理人员有 A、B、C、D 四人。某天 A 因为集群出了问题，临时改了集群配置，但没有及时通知 B、C、D。过了一个月 B、C、D 三人才突然发现配置已发生变更，然后询问到 A，发现是他做的操作。但是 A 已经记不得当时他具体改的是什么配置项了。这就会造成配置变更记录的丢失。

以上两点问题主要在集群规模比较大的时候容易显现出来。

7.6.2 现有配置管理工具

针对以上问题，目前是否有现成的工具能够帮助我们解决配置管理的问题呢？答案是有的。目前比较流行的工具有 2 个：Chef 和 Puppet。这 2 个工具笔者都使用过，但是 Chef 会更加熟悉一些。所以这里主要分享笔者使用 Chef 做集群配置管理的一些体会。

第一点，学习成本较高。这些开源工具并不是拿来立即就能使用的，用户至少需要了解开源工具的操作流程，同时最好也要了解它的相关原理设计。

第二点，需要投入时间去学习与使用。笔者在使用 Chef 做集群配置管理的时候，是看中了它的历史版本管理的功能以及动态生成配置的功能。但是如果你想很好地使用它，还是需要投入一些时间进去，而且要不断地去实践。否则，你可能只是仅仅用到了它的一小部分功能而已，还不如我们写个脚本批处理程序执行一遍来得快。

总而言之，对于开源工具的使用，笔者认为不在于它本身是多么完美、好用，而是在于用户是否能够花时间、人力去把它用好。用好了，自然就能发挥它的功能，如果用不好，那还是建议用我们更加熟悉的方式来做，例如自己开发一套配置管理系统。

7.6.3 运用 Git 来做配置管理

有什么办法可以让我们以最低的成本来做配置管理这件事情呢？我们马上可以联想到 Git。因为 Git 对于我们来说非常熟悉，而且它与 Svn 类似，可以做项目的管理。所以在这节中，我们尝试使用 Git 来做集群配置管理的解决方案。由于 Git 本身的特点，它可以直接帮助我们解决配置记录变更的问题，通过 `git log` 命令能找到所有的提交记录。但是另外一个问题，也就是配置的差异管理，这个需要我们自己想办法进行解决。

假设我们已经在 Git 上创建好了一个工程，里面的内容就是待管理的配置文件。在这

种情况下我们如何做到配置的差异化管理呢？首先一点，我们需要明白会存在哪些差异点，然后再对配置的目录结构做调整。在前面的内容中，我们提到了其中一个差异点：机型的差异。其实如果把考虑范围放大，还会有如下的差异点：

- 集群归属差异。可能你所维护的集群不止一个，并且不同集群分布在不同的机房内，这里就会有集群本身的差异。
- 版本的差异。如果集群未来会进行升级操作，势必会带来版本的变化。新老版本的配置之间会存在一定的差别。

结合以上提出的 3 点差异性，我们可以对 Git 配置项工程进行如下的设计：

```
branch:clusterA——-/versions/2.7.1/64G/etc
```

对于集群自身的不同，我们可以以 branch 分支做区分。然后是 versions 版本目录，在此目录下是针对多个版本的配置项，如果 Hadoop 以后发布了 2.8.0 版本，我们就可以加在此目录下。接下来是机型的差异，机型的差异通过 64G、128G 等带有特征性的名称做区分，这里的 64G 指的是 64GB 内存的节点。在机型的子目录中，才是最终存放的目录。这里 etc 目录实质上就是 Hadoop 安装包下的 etc 目录。

配置项过程设计好了，我们如何将它运用到实际的工作中去呢？直接放到 Hadoop 安装包下？这样显然是不行的，因为目录结构已经完全不同了。所以解决的办法是通过软链接的形式。将节点中 Hadoop 安装包中的 etc 目录软链接到上面具体机型下的 etc 目录，效果如下：

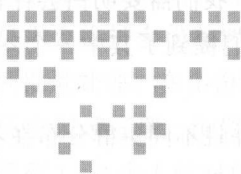
```
hadoop-current/etc -> /versions/2.7.1/64G/etc
```

当我们想要做配置的变更时，只要在本地上机器上修改这个 Git 工程项目，进行提交，然后在集群中的某台机器上做 git pull 批量更新动作即可。以后我们的运维人员就不需要再登录机器去改配置了。

本节所述的配置管理方案算不上是什么复杂的方案，主要还是与大家讨论一下在集群维护中的配置管理问题，以及我们是否有好的办法去解决它，希望能够给大家带来一些启发性的思考。

7.7 小结

本章讲述了关于 HDFS 数据管理的多个实践方案。不同方案之间由于难度不相同，所花的篇幅也差别较大。其中 HDFS 的数据规模监控在实际运维中是比较重要的一点，这样的数据对于我们进行集群规模的扩容具有十分重要的参考意义。而 HDFS 的数据迁移方案无疑是本章的一个难点，如何做到数据的准实时同步是其中一个非常关键的步骤。



HDFS 的数据读写

本章我们将会介绍两个局部的改造。与前一章不同，本章的改造内容需要基于 Hadoop 源代码进行，改造的目标对象都是磁盘。因为 HDFS 每天的读写操作都会经过磁盘，所以对其进行优化、改造也是必不可少的。本章将会介绍两方面的改造，第一，磁盘选择策略的改造，目前是采用轮询的方式进行磁盘的选择。第二，“慢磁盘”的监控改造，此监控能够让我们马上发现“慢磁盘”，并将其从节点中进行移除。

8.1 DataNode 引用计数磁盘选择策略

DataNode 引用计数磁盘选择策略是一种基于引用计数值作为衡量指标的磁盘选择策略。这里的“引用计数”指的是当前磁盘被引用的计数，意思是指当前磁盘进行读写操作的次数。磁盘引用计数很高表明此盘目前正在被大量地读写。所以在一定程度上而言，磁盘引用计数策略考虑的因素在于磁盘 IO 的负载而非磁盘间的数据平衡情况。此策略类在目前 HDFS 中并不存在，是笔者自定义的一个策略类。所以在本节内容的讲述中，笔者将会详细地讲述此策略类与现有策略类的异同以及它们各自适用的场景。

8.1.1 HDFS 现有磁盘选择策略

随着节点数的扩增，集群中的总磁盘数也会跟着线性变化，这么多块磁盘，会造成一个问题：数据不平衡现象。这是很容易发生的，原因有以下 2 点：

- 1) HDFS 写操作不当导致。

2) 新老机器上线使用时间不同, 造成新机器数据少, 老机器数据多的问题。

第二点通过 Balancer 操作可以解决, 第一个问题才是最根本的。为了解决磁盘数据空间不均衡的问题, HDFS 目前的两套磁盘选择策略都是围绕着“数据平衡”的目标设计的。下面主要介绍这 2 个磁盘选择策略。

1. RoundRobinVolumeChoosingPolicy

标题所示的类名称可以拆成 2 个单词: RoundRobin 和 VolumeChoosingPolicy。VolumeChoosingPolicy 理解为磁盘选择策略, RoundRobin 则是一个专业术语, 叫做“轮询”。类似的还有一些别的术语: Round-Robin Scheduling (轮询调度)、Round-Robin 算法等等。一句话来概括“轮询”的意思: 一个一个地去遍历, 到尾了, 再从头开始。图 8-1 为轮询算法原理。

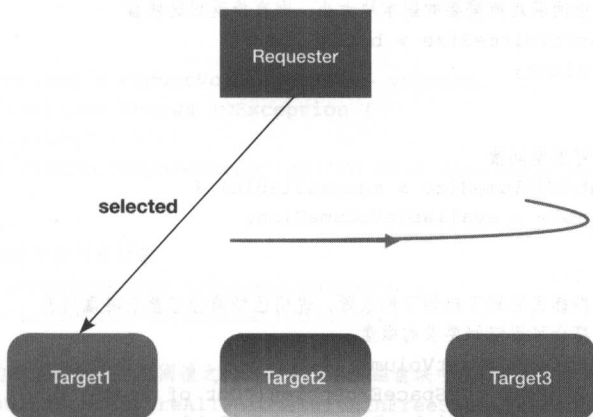


图 8-1 RoundRobin 原理图

下面给出此策略的核心代码, 如下所示:

```
// 轮询方式的磁盘选择策略类
public class RoundRobinVolumeChoosingPolicy<V extends FsVolumeSpi>
    implements VolumeChoosingPolicy<V> {
    public static final Log LOG = LogFactory.getLog(RoundRobinVolumeChoosingPolicy.class);

    private int curVolume = 0;

    @Override
    public synchronized V chooseVolume(final List<V> volumes, long blockSize)
        throws IOException {

        // 如果磁盘数目小于 1 个, 则抛异常
        if(volumes.size() < 1) {
            throw new DiskOutOfSpaceException("No more available volumes");
        }
    }
}
```

```

// 如果由于失败磁盘导致当前磁盘下标越界了,则将下标置为 0
if(curVolume >= volumes.size()) {
    curVolume = 0;
}

// 起始下标赋值
int startVolume = curVolume;
long maxAvailable = 0;

while (true) {
    // 获取当前下标所代表的磁盘
    final V volume = volumes.get(curVolume);
    // 下标递增
    curVolume = (curVolume + 1) % volumes.size();
    // 获取当前选中磁盘的可用剩余空间
    long availableVolumeSize = volume.getAvailable();
    // 如果可用空间满足所需要的副本块大小,则直接返回这块盘
    if (availableVolumeSize > blockSize) {
        return volume;
    }

    // 更新最大可用空间值
    if (availableVolumeSize > maxAvailable) {
        maxAvailable = availableVolumeSize;
    }

    // 如果当前指标又回到了起始下标位置,说明已经遍历完整个磁盘列表
    // 没有找到符合可用空间要求的磁盘
    if (curVolume == startVolume) {
        throw new DiskOutOfSpaceException("Out of space: "
            + "The volume with the most available space (= " + maxAvailable
            + " B) is less than the block size (= " + blockSize + " B).");
    }
}
}
}

```

理论上来说这种策略是比较符合数据平衡的目标的,因为一个个地写每块盘,写入的次数都差不多,不存在哪块盘多写少写的现象。但是唯一的不足之处在于每次写入的数据量是无法控制的。可能我某次操作在 A 盘上写入了 512 字节的数据,在轮到 B 盘写的时候我写了 128MB 的数据,数据就不均衡了。所以说轮询策略在某种程度上来说是理论上均衡,但还不是最好的,更好的是下面介绍的策略。

2. AvailableSpaceVolumeChoosingPolicy

剩余可用空间磁盘选择策略。这个磁盘选择策略比第一种策略就精妙很多了。首先它根据一个阈值,将所有的磁盘分为了两大类:高可用空间磁盘列表和低可用空间磁盘列表。然后通过一个随机数概率,会以比较高的概率选择高剩余空间磁盘列表中的磁盘,然后对

这些磁盘列表使用轮询策略进行选择，下面是相关代码：

```
// 根据剩余可用空间进行优先选择的磁盘选择策略类
public class AvailableSpaceVolumeChoosingPolicy<V extends FsVolumeSpi>
    implements VolumeChoosingPolicy<V>, Configurable {

    ...
    // 用于一般的需要平衡磁盘的轮询磁盘选择策略
    private final VolumeChoosingPolicy<V> roundRobinPolicyBalanced =
        new RoundRobinVolumeChoosingPolicy<V>();
    // 用于可用空间高的磁盘的轮询磁盘选择策略
    private final VolumeChoosingPolicy<V> roundRobinPolicyHighAvailable =
        new RoundRobinVolumeChoosingPolicy<V>();
    // 用于可用空间低的磁盘的轮询磁盘选择策略
    private final VolumeChoosingPolicy<V> roundRobinPolicyLowAvailable =
        new RoundRobinVolumeChoosingPolicy<V>();

    @Override
    public synchronized V chooseVolume(List<V> volumes,
        long replicaSize) throws IOException {
        if (volumes.size() < 1) {
            throw new DiskOutOfSpaceException("No more available volumes");
        }

        // 获取所有磁盘包装列表对象
        AvailableSpaceVolumeList volumesWithSpaces =
            new AvailableSpaceVolumeList(volumes);

        // 如果所有的磁盘在数据平衡阈值之内，则在所有的磁盘块中直接进行轮询选择
        if (volumesWithSpaces.areAllVolumesWithinFreeSpaceThreshold()) {
            V volume = roundRobinPolicyBalanced.chooseVolume(volumes, replicaSize);
            if (LOG.isDebugEnabled()) {
                LOG.debug("All volumes are within the configured free space balance " +
                    "threshold. Selecting " + volume + " for write of block size " +
                    replicaSize);
            }
            return volume;
        } else {
            V volume = null;
            // 如果存在数据不平衡的现象，则从低剩余空间磁盘块中选出可用空间最大值
            long mostAvailableAmongLowVolumes = volumesWithSpaces
                .getMostAvailableSpaceAmongVolumesWithLowAvailableSpace();

            // 得到高可用空间磁盘列表
            List<V> highAvailableVolumes = extractVolumesFromPairs(
                volumesWithSpaces.getVolumesWithHighAvailableSpace());
            // 得到低可用空间磁盘列表
            List<V> lowAvailableVolumes = extractVolumesFromPairs(
                volumesWithSpaces.getVolumesWithLowAvailableSpace());

            float preferencePercentScaler =
```

```

        (highAvailableVolumes.size() * balancedPreferencePercent) +
        (lowAvailableVolumes.size() * (1 - balancedPreferencePercent));
// 计算平衡比值, balancedPreferencePercent 越大, highAvailableVolumes.size() 所
// 占的比重会变大
// 整个比例值也会变大, 就会有更高的随机概率在这个值下
float scaledPreferencePercent =
    (highAvailableVolumes.size() * balancedPreferencePercent) /
    preferencePercentScaler;
// 如果低可用空间磁盘列表中最大的可用空间无法满足副本大小
// 或随机概率小于比例值, 就在高可用空间磁盘中进行轮询调度选择
if (mostAvailableAmongLowVolumes < replicaSize ||
    random.nextFloat() < scaledPreferencePercent) {
    volume = roundRobinPolicyHighAvailable.chooseVolume(
        highAvailableVolumes, replicaSize);
    if (LOG.isDebugEnabled()) {
        LOG.debug("Volumes are imbalanced. Selecting " + volume +
            " from high available space volumes for write of block size "
            + replicaSize);
    }
} else {
    // 否则在低磁盘空间列表中选择磁盘
    volume = roundRobinPolicyLowAvailable.chooseVolume(
        lowAvailableVolumes, replicaSize);
    if (LOG.isDebugEnabled()) {
        LOG.debug("Volumes are imbalanced. Selecting " + volume +
            " from low available space volumes for write of block size "
            + replicaSize);
    }
}
return volume;
}
}

```

低剩余空间磁盘和高剩余空间磁盘的标准是这样定义的:

```

// 获取低剩余空间磁盘列表
public List<AvailableSpaceVolumePair> getVolumesWithLowAvailableSpace() {
    long leastAvailable = getLeastAvailableSpace();
    List<AvailableSpaceVolumePair> ret = new ArrayList<AvailableSpaceVolumePair>();
    for (AvailableSpaceVolumePair volume : volumes) {
        // 如果当前磁盘可用空间小于最小空间与平衡空间阈值的和,
        // 则将此磁盘加入低磁盘空间列表中
        if (volume.getAvailable() <= leastAvailable + balancedSpaceThreshold) {
            ret.add(volume);
        }
    }
    return ret;
}

// 获取高剩余空间列表
public List<AvailableSpaceVolumePair> getVolumesWithHighAvailableSpace() {

```

```

long leastAvailable = getLeastAvailableSpace();
List<AvailableSpaceVolumePair> ret = new ArrayList<AvailableSpaceVolumePair>();
for (AvailableSpaceVolumePair volume : volumes) {
    // 高剩余空间磁盘选择条件与上面相反
    if (volume.getAvailable() > leastAvailable + balancedSpaceThreshold) {
        ret.add(volume);
    }
}
return ret;
}

```

到此我们已经了解了 HDFS 目前现有的两种磁盘选择策略。那么 HDFS 在使用这些策略上到底是不是完美的呢？答案显然不是。下面是笔者总结出的两点不足之处：

- HDFS 默认的磁盘选择策略是 RoundRobinVolumeChoosingPolicy，而不是更优的 AvailableSpaceVolumeChoosingPolicy。笔者猜测的原因是 AvailableSpaceVolumeChoosingPolicy 是后来才有的，但是默认值的选择没有改，依然是老的策略。但同时也需要指出 AvailableSpaceVolumeChoosingPolicy 策略可能存在性能问题。一旦某个节点新添加了若干块新磁盘，按照它的策略，大部分的数据将会被写入到这些新盘上。大量的数据同时写入磁盘会造成不小的性能影响。
- 磁盘选择策略考虑的因素过于单一。磁盘可用空间只是其中一个因素，其实还有别的指标也值得参考，比如当前磁盘的 IO 情况。当我们准备写数据到磁盘的时候，我们当然希望找一些闲置的磁盘进行数据的写入。对于繁忙的磁盘而言，这只会更加影响它的写入速度。此因素也是下面笔者自定义的新磁盘选择策略的一个根本需求点。

8.1.2 自定义磁盘选择策略

寻找自定义磁盘选择策略的根本依赖点在于 ReferenceCount（引用计数），它能让你知道有多少对象正在操作当前磁盘。引用计数的原理在很多地方都有用到，比如 JVM 中通过对象是否还有外部引用来判断是否能够进行垃圾回收。在磁盘相关类 FsVolume 中恰好有一个引用计数相关的变量，如下：

```

public class FsVolumeImpl implements FsVolumeSpi {
    ...
    private CloseableReferenceCount reference = new CloseableReferenceCount();
}

```

这里我们需要将此变量值开放出去，便于我们的调用：

```

// 获取当前磁盘的引用计数
public int getReferenceCount() {
    return this.reference.getReferenceCount();
}

```

然后模仿 AvailableSpaceVolumeChoosingPolicy 策略进行选择，核心代码如下：

```
@Override
public synchronized V chooseVolume(final List<V> volumes, long blockSize)
    throws IOException {

    if (volumes.size() < 1) {
        throw new DiskOutOfSpaceException("No more available volumes");
    }

    V volume = null;

    // 获取当前磁盘中引用次数最少的一块盘
    int minReferenceCount = getMinReferenceCountOfVolumes(volumes);
    // 根据最少引用次数以及引用计数临界值得到低引用计数磁盘列表
    List<V> lowReferencesVolumes =
        getLowReferencesCountVolume(volumes, minReferenceCount);
    // 根据最少引用次数以及引用计数临界值得到高引用计数磁盘列表
    List<V> highReferencesVolumes =
        getHighReferencesCountVolume(volumes, minReferenceCount);

    // 判断低引用磁盘列表中是否存在满足要求块大小的磁盘，如果有，优先从低引用计数磁盘中进行轮询
    // 磁盘的选择
    if (isExistVolumeHasFreeSpaceForBlock(lowReferencesVolumes, blockSize)) {
        volume =
            roundRobinPolicyLowReferences.chooseVolume(lowReferencesVolumes,
                blockSize);
    } else {
        // 如果低引用计数磁盘块中没有可用空间的块，则再从高引用计数的磁盘列表中进行磁盘的选择
        volume =
            roundRobinPolicyHighReferences.chooseVolume(highReferencesVolumes,
                blockSize);
    }

    return volume;
}
```

附上相应的单元测试，测试已经通过。大家可以自行在自己的环境中进行测试。

```
// 引用计数磁盘选择策略模拟测试
@Test
public void testReferenceCountVolumeChoosingPolicy() throws Exception {
    @SuppressWarnings("unchecked")
    final ReferenceCountVolumeChoosingPolicy<FsVolumeSpi> policy =
        ReflectionUtils.newInstance(ReferenceCountVolumeChoosingPolicy.class,
            null);
    // 初始化引用计数磁盘选择策略
    initPolicy(policy);
    final List<FsVolumeSpi> volumes = new ArrayList<FsVolumeSpi>();

    // 增加2块低引用计数的磁盘
```

```

// 第一块盘, 只被引用了1次的磁盘
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(0).getReferenceCount()).thenReturn(1);
Mockito.when(volumes.get(0).getAvailable()).thenReturn(100L);

// 第二块盘, 被引用了2次的磁盘
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(1).getReferenceCount()).thenReturn(2);
Mockito.when(volumes.get(1).getAvailable()).thenReturn(100L);

// 增加2块高引用计数的磁盘
// 第三块盘, 被引用了4次的磁盘
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(2).getReferenceCount()).thenReturn(4);
Mockito.when(volumes.get(2).getAvailable()).thenReturn(100L);

// 第四块盘, 被引用了5次的磁盘
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(3).getReferenceCount()).thenReturn(5);
Mockito.when(volumes.get(3).getAvailable()).thenReturn(100L);

// 用引用计数磁盘选择策略选择一个可用空间至少50字节以上的盘,
// 于是将会选中第一块盘, 因为引用次数少, 而且可用空间满足条件
Assert.assertEquals(volumes.get(0), policy.chooseVolume(volumes, 50));

volumes.clear();

// 下面测试的场景是当低引用计数的磁盘可用空间不足的情况

// 增加第一块盘, 引用计数1, 可用空间50字节
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(0).getReferenceCount()).thenReturn(1);
Mockito.when(volumes.get(0).getAvailable()).thenReturn(50L);

// 增加第二块盘, 引用计数2, 可用空间50字节
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(1).getReferenceCount()).thenReturn(2);
Mockito.when(volumes.get(1).getAvailable()).thenReturn(50L);

// 增加第三块盘, 引用计数4, 可用空间200字节
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(2).getReferenceCount()).thenReturn(4);
Mockito.when(volumes.get(2).getAvailable()).thenReturn(200L);

// 增加第三块盘, 引用计数5, 可用空间200字节
volumes.add(Mockito.mock(FsVolumeSpi.class));
Mockito.when(volumes.get(3).getReferenceCount()).thenReturn(5);
Mockito.when(volumes.get(3).getAvailable()).thenReturn(200L);

// 用引用计数磁盘选择策略选择一个可用空间至少在100字节以上的盘,
// 尽管引用磁盘1、2引用计数少, 但是其可用空间不足, 退而求其次, 只能选择

```

```
// 第三块盘。  
Assert.assertEquals(volumes.get(2), policy.chooseVolume(volumes, 100));  
}
```

当然引用计数磁盘选择策略也不见得是最好的，因为这里忽略了磁盘间数据不平衡的问题。这个弊端会慢慢凸显出来，所以说很难有一个策略是绝对完美的。最好的办法是根据用户使用场景使用最合适的磁盘选择策略，比如定期更换策略以此达到最佳的效果。引用计数磁盘选择策略的相关代码可以从下面笔者的 GitHub 链接中查阅、学习，链接如下：<https://github.com/linyiqun/open-source-patch/blob/master/hdfs/others/HDFS-volume-ChoosingPolicy/HDFS-001.patch>。

8.2 Hadoop 节点“慢磁盘”监控

Hadoop 节点“慢磁盘”监控源自于笔者工作中的一次“慢磁盘”故障处理，在问题的解决中实现了“慢磁盘”的监控。这里的“慢磁盘”指的是写入数据非常慢的一类磁盘。其实慢磁盘并不少见，当机器运行时间长了，上面跑的任务多了，磁盘的读写性能自然会退化，严重时就会出现写入数据延时的问题。目前磁盘监控在 HDFS 中并没有做得很全，大多数都是对 DataNode 整体做监控，可以说这是一个盲区。本节将完整地介绍笔者在工作中解决此次慢磁盘问题的整个过程，主要包括慢磁盘的发现和慢磁盘的监控两方面内容。

8.2.1 慢磁盘的定义以及如何发现

在这里笔者暂且用“慢磁盘”来解释这个现象，英文描述为 slow-writed disk，译为写入操作很慢的磁盘。这里的写操作主要包括创建文件、目录，写文件数据等操作。而慢磁盘指的是写操作耗时远远超出平均时间的一类磁盘。笔者在工作中就碰到了这样的场景，其他正常的盘基本上创建 1 个 test 目录，只需 1/10 或者快的 1/100 秒左右的时间。而笔者惊奇地发现有块盘竟然花了 5 分钟左右，更奇怪的是，这个现象会时不时地出现，并不是每次都有。一旦出现了慢磁盘，将会严重拖慢这个节点的整体运行效率，继而让此节点成为集群中的慢节点，最后影响整个集群。那么问题来了，既然慢磁盘这么重要，我们怎么准确定位到是哪台机器的哪块磁盘有问题呢？集群中包含了那么多个节点，每个节点上又有那么多块盘，一个个地去找绝对不是一个合适的办法。

下面教大家几个方法来发现慢磁盘：

- ❑ 通过心跳未联系时间。一般如果出现慢磁盘现象，会影响到 DataNode 与 NameNode 之间的心跳，图 8-2 中的 Last contact 值会持续变大，这个值为此 DataNode 与 NameNode 之间目前失去联系的时间（时间按秒算）。正常情况下，Last contact 值应小于 3，因为 DataNode 心跳的默认发送间隔是 3 秒。

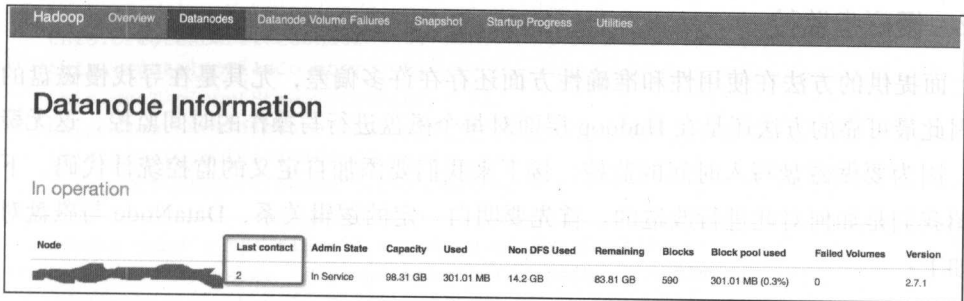


图 8-2 DataNode 与 NameNode 的心跳未联系间隔

□ 通过 Ganglia 对 DataNode 写操作的相关监控，这个是传统的方式，如图 8-3 所示。

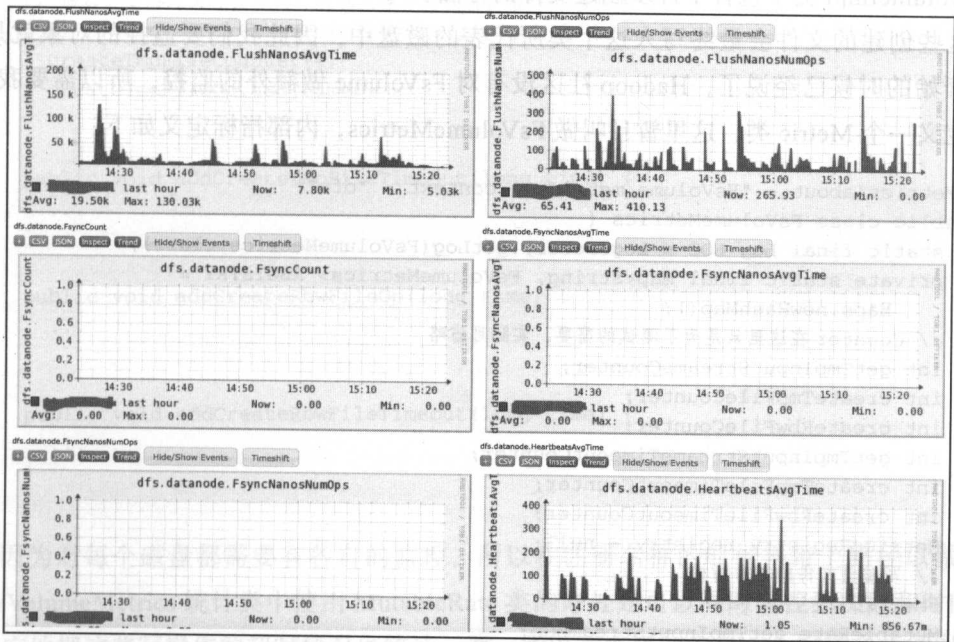


图 8-3 Ganglia 对 DataNode 写操作的相关监控

对比几个特殊的节点，观察这些节点在写操作时间上有没有特别长的，使用这种方法可以确定可疑慢磁盘所在的节点。假设异常节点已经发现，可以使用最简单的方法来发现节点上的慢磁盘，即写一个脚本在所有的磁盘上执行。

```
time mkdir test
rm -r -f test
```

观察哪个磁盘所花的时间最长就可以了。我们也可以用 Linux 系统中专门检查磁盘读写性能的命令，这种方式会更加准确一些。

8.2.2 慢磁盘监控

上面提供的方法在使用性和准确性方面还存在许多偏差，尤其是在寻找慢磁盘的方法上。因此最可靠的方法还是在 Hadoop 层面对每个磁盘进行写操作的时间监控，这无疑是最准的。因为要做磁盘写入时间的监控，接下来我们要添加自定义的监控统计代码。下面简单介绍我们是如何对此进行改造的。首先要明白一定的逻辑关系，DataNode 与磁盘对应的关系如下：

```
DataNode-->FsDatasetImpl-->volumesList
```

在 volumesList 里包含了各个磁盘上的数据存储空间。每个磁盘对应的类是 FsVolumeImpl，在 FsVolumeImpl 类中包含了许多创建文件的方法。

这些创建的文件会最终写入这个类所代表的磁盘中，因此我们要监控的对象就是它。本节开始的时候已经说了，Hadoop 社区没有对 FsVolume 做额外的监控，所以需要我们自己新定义一个 Metric 类，这里暂且叫做 FsVolumeMetrics，内部指标定义如下：

```
@Metrics(about = "FsVolume metrics", context = "dfs")
public class FsVolumeMetrics {
    static final Log LOG = LogFactory.getLog(FsVolumeMetrics.class);
    private static final Map<String, FsVolumeMetrics> REGISTRY =
        Maps.newHashMap();
    // Counter 在这里只是为了测试的需要，实际可省略
    int getTmpInputStreamsCounter;
    int createTmpFileCounter;
    int createRbwFileCounter;
    int getTmpInputStreamsTimeoutCounter;
    int createTmpFileTimeoutCounter;
    int createRbwFileTimeoutCounter;
    MetricsRegistry registry = null;
    // 写磁盘文件的耗时指标
    @Metric
    MutableRate getTmpInputStreamsOp;
    @Metric
    MutableRate createTmpFileOp;
    @Metric
    MutableRate createRbwFileOp;

    // 写磁盘文件的超时记录指标
    @Metric
    MutableRate getTmpInputStreamsTimeout;
    @Metric
    MutableRate createTmpFileTimeout;
    @Metric
    MutableRate createRbwFileTimeout;

    // 初始化磁盘 Metric 统计类，将各个计数值置为 0
```

```

private FsVolumeMetrics(FsVolumeImpl volume) {
    this.createRbwFileCounter = 0;
    this.createTmpFileCounter = 0;
    this.getTmpInputStreamsCounter = 0;
    this.createRbwFileTimeoutCounter = 0;
    this.createTmpFileTimeoutCounter = 0;
    this.getTmpInputStreamsTimeoutCounter = 0;

    String name = "fsVolume:" + volume.getBasePath();
    LOG.info("Register fsVolume metric for path: " + name);
    registry = new MetricsRegistry(name);
}

...
// 增加创建临时文件操作的计数以及操作的耗时，后续方法同理
public void addCreateTmpFileOp(long time) {
    createTmpFileCounter++;
    createTmpFileOp.add(time);
}

public void addCreateTmpFileTimeout(long time) {
    ...
}

public void addCreateRbwFileOp(long time) {
    ...
}

public void addCreateRbwFileTimeout(long time) {
    ...
}
}

```

因为对每个磁盘都需要有各自的监控，所以在注册名称的时候得带上路径以做区分。在 `FsVolumeMetrics` 统计类中使用 `MutableRate` 类的好处是可以同时监控到次数和时间，然后对应地把这些写操作的超时情况也统计一遍。所以这里要新定义一个写磁盘的超时时间，如下所示：

```

public static final String DFS_WRITE_VOLUME_THRESHOLD_TIME_MS =
    "dfs.write.volume.threshold.time.ms";
public static final long DFS_WRITE_VOLUME_THRESHOLD_TIME_MS_DEFAULT = 300;

```

接下来是注册此 Metric 类代码，注意这需要在 `FsVolumeImpl` 类中注册：

```

FsVolumeImpl(FsDatasetImpl dataset, String storageID, File currentDir,
    Configuration conf, StorageType storageType) throws IOException {
    this.dataset = dataset;
    this.storageID = storageID;
    this.reserved = conf.getLong(

```

```

        DFSConfigKeys.DFS_DATANODE_DU_RESERVED_KEY,
        DFSConfigKeys.DFS_DATANODE_DU_RESERVED_DEFAULT);
this.reservedForRbw = new AtomicLong(0L);
....
// 在 FsVolumeImpl 构造方法中进行 Metric 类的初始化
metric = FsVolumeMetrics.create(this);
}

```

这样每个磁盘就会有各自的监控类了。然后进行写操作耗时的监控，这里给出对其中 createRbw 方法的监控，其余方法类似，就不列举了，在本节末尾会给出代码链接。

```

@Override // FsDatasetSpi
public synchronized ReplicaHandler createRbw(
    StorageType storageType, ExtendedBlock b, boolean allowLazyPersist)
    throws IOException {
    ReplicaInfo replicaInfo = volumeMap.get(b.getBlockPoolId(),
        b.getBlockId());
    ....
}
FsVolumeImpl v = (FsVolumeImpl) ref.getVolume();
// create an rbw file to hold block in the designated volume
File f;
try {
    // 进行写操作前后时间记录
    long startTime = Time.monotonicNow();
    f = v.createRbwFile(b.getBlockPoolId(), b.getLocalBlock());
    long duration = Time.monotonicNow() - startTime;
    // 如果写操作时间超出阈值，则进行计数递增操作
    if (duration > volumeThresholdTime) {
        LOG.warn("Slow create RbwFile to volume=" + v.getBasePath() + " took "
            + duration + "ms (threshold=" + volumeThresholdTime + "ms)");
        v.metric.addCreateRbwFileTimeout(duration);
    }
    v.metric.addCreateRbwFileOp(duration);
} catch (IOException e) {
    IOUtils.cleanup(null, ref);
    throw e;
}
....
}

```

代码是加在 FsDatasetImpl 类中的，因为方法是在这发起的。所以总的来说，监控处理的代码逻辑其实并不复杂。既然是 Hadoop 内部的统计监控类，它能够直接展示在 Ganglia 的界面上，见图 8-4。

因为笔者配置的数据目录是 /home/data/data/hadoop/dfs/data，所以图 8-4 中会出现上面那么长的图形小标题。图 8-4 就是我们想要达到的最终效果，希望能带给大家收获。此功能笔者已经打成 patch，提交开源社区，编号 HDFS-9510。要使用此功能的同学可以自行获

取代码，HDFS-9510 地址如下：

<https://issues.apache.org/jira/browse/HDFS-9510>。

如果慢磁盘已经发现了，怎么解决呢？最干脆的办法就是立即下线，不要再往这块盘上写数据了，并联系运维部门进行处理或者说自己内部想办法解决。但是还是那句话，像慢磁盘这样的偏硬件性的问题还是交给这方面专业的人去解决比较稳妥。

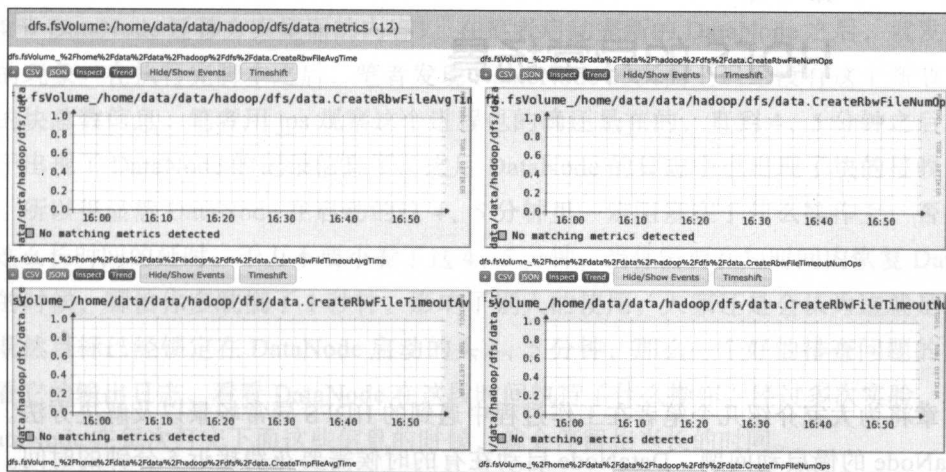


图 8-4 Ganglia 的 FsVolume 写磁盘操作监控

8.3 小结

本章介绍的两个基于磁盘的改造更多的是一种创新的改造，难度并不会太大，大家只需理解其用意本身即可。本章提出的“慢磁盘”的监控还是比较重要的，往往有的时候这类盘会拖慢一个节点。

HDFS 的异常场景

本章将为大家介绍几个笔者在工作过程中遇到的 HDFS 异常场景以及解决方法。首先是 DataNode 的慢启动问题，DataNode 启动在有的时候需要花费接近 5 分钟的时间。其次是 Hadoop 节点在中止下线操作后，会出现大量多余块的情况。最后是 HDFS 在读写数据时发生的线程泄露问题，线程泄露类似于内存泄露问题，是一个比较严重的问题。

9.1 DataNode 慢启动问题

正常情况下，一个 DataNode 启动的速度一般在几秒或者十几秒的时间。但是在某些特殊的场合下，DataNode 会出现启动长达几分钟的现象，在本节中我们称此现象为“DataNode 慢启动”。DataNode 的慢启动不会导致它上面的数据发生丢失，但是它会影响到服务的恢复。假设集群中所有的 DataNode 都出现了长达 4、5 分钟的慢启动现象，那么集群对外提供服务的时间就会受到影响。本节将完整地讲述 DataNode 慢启动的问题，包括慢启动现象的发现、慢启动的原因分析和最后的问题解决。

9.1.1 DataNode 慢启动现象

首先看到这个小标题，可能有人会有疑问：DataNode 还会出现慢启动现象？DataNode 执行了 `sbin/hadoop-daemon.sh start datanode` 命令后不是几秒钟的事情吗？这当然没有错，在绝大多数的场景下，DataNode 的启动就是简单的这么几个步骤。但是不知道大家有没有尝试过如下的情况：

- 1) 停止机器上的 DataNode 服务。
- 2) 将此节点进行机房搬迁, 搬迁后此节点将会拥有新的主机名和 ip 地址。
- 3) 在第二步骤的搬迁过程中耗费了 20、30 分钟甚至长达数小时。
- 4) 重启被更换掉主机名、ip 地址的 DataNode。

笔者在最近一段时间的 DataNode 迁移中就遇到了上述的场景(感兴趣的同学可以查阅第 7.4 节 DataNode 迁移方案里面的内容)。在笔者启动完新的 DataNode 之后, 就发生了慢启动的现象。在执行完脚本之后, 笔者发现 NameNode 的页面上迟迟没有这个新节点汇报上来的块总数信息。笔者用 jps 观察这个进程也的确还是在的, 直到 4、5 分钟之后, 页面上终于出现了 DataNode 的记录信息了。之后 DataNode 的日志中也出现了块的接收、删除记录。所以很显然 DataNode 在启动的这 4、5 分钟里一定阻塞在了什么操作上, 否则不会出现这么长时间的延时。千万不要小看了这 4、5 分钟, 当你需要在短时间内恢复 DataNode 服务的时候, 哪怕你多耽搁了 1 秒钟, 影响了别人的使用, 人家还是会认为这就是你的问题。既然目标已经锁定在 DataNode 启动的头 4、5 分钟, 那么一个好的排查问题的办法就是去看它的输出日志, 看看 DataNode 在这段时间执行了什么操作。经过多次实验, 笔者发现 DataNode 在每次打完下面这些信息的时候, 就会停留相当长的时间。

```
2016-01-06 16:05:08,181 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Scanning block pool BP-1942012336-xx.xx.xx.xx-1406726500544
on volume /home/data/data/hadoop/dfs/data/data10/current...
2016-01-06 16:05:08,181 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Scanning block pool BP-1942012336-xx.xx.xx.xx-1406726500544
on volume /home/data/data/hadoop/dfs/data/data11/current...
2016-01-06 16:05:08,181 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Scanning block pool BP-1942012336-xx.xx.xx.xx-1406726500544
on volume /home/data/data/hadoop/dfs/data/data12/current...
2016-01-06 16:09:49,646 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Time taken to scan block pool BP-1942012336-xx.xx.xx.xx-
1406726500544 on /home/data/data/hadoop/dfs/data/data7/current: 281466ms
2016-01-06 16:09:54,235 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Time taken to scan block pool BP-1942012336-xx.xx.xx.xx-
1406726500544 on /home/data/data/hadoop/dfs/data/data9/current: 286054ms
2016-01-06 16:09:57,859 INFO org.apache.hadoop.hdfs.server.datanode.fsdataset.
impl.FsDatasetImpl: Time taken to scan block pool BP-1942012336-xx.xx.xx.xx-
1406726500544 on /home/data/data/hadoop/dfs/data/data2/current: 289680ms
```

我们可以从日志中看到, 在 DataNode 添加完磁盘块后, 进行扫描操作的时候, 从 16:05 分直接跳到了 16:09 的记录, 就是在最后一次 Scanning block 的那行。显然就是这段时间导致的 DataNode 慢启动。所以我们可以以此作为关键线索, 进行分析, 一个有效的办法是通过日志进行代码追踪分析。

9.1.2 代码追踪分析

通过上文日志记录信息的内容显示，这条信息是 `FsDatasetImpl` 中的 `Log` 对象打出的，但是这个 `Log` 对象其实是在另外一个叫 `FsVolumeList` 的类中被调用的，代码如下：

```
void addBlockPool(final String bpid, final Configuration conf) throws IOException {
    long totalStartTime = Time.monotonicNow();

    final List<IOException> exceptions = Collections.synchronizedList(
        new ArrayList<IOException>());
    List<Thread> blockPoolAddingThreads = new ArrayList<Thread>();
    for (final FsVolumeImpl v : volumes.get()) {
        Thread t = new Thread() {
            public void run() {
                try (FsVolumeReference ref = v.obtainReference()) {
                    FsDatasetImpl.LOG.info("Scanning block pool " + bpid +
                        " on volume " + v + "...");
                    // 在引入新磁盘前记录起始时间
                    long startTime = Time.monotonicNow();
                    // 执行添加动作
                    v.addBlockPool(bpid, conf);
                    // 记录操作结束时间
                    long timeTaken = Time.monotonicNow() - startTime;
                    FsDatasetImpl.LOG.info("Time taken to scan block pool " + bpid +
                        " on " + v + ": " + timeTaken + "ms");
                } catch (ClosedChannelException e) {
                    ...
                }
            };
            blockPoolAddingThreads.add(t);
            t.start();
        }
        for (Thread t : blockPoolAddingThreads) {
            try {
                // 调用 join 方法等待此线程运行结束
                t.join();
            } catch (InterruptedException ie) {
                throw new IOException(ie);
            }
        }
    }
    ...
}
```

在添加每块盘上的 `BlockPool` 操作的时候，是采用创建线程的方式，并且调用了 `join` 方法等待线程结束。我们再查看其中的操作来看看到底是在干什么，首先查看下面的这个方法：

```
void addBlockPool(String bpid, Configuration conf) throws IOException {
    File bpdire = new File(currentDir, bpid);
```



```
// 新建 BlockPoolSlice 对象
BlockPoolSlice bp = new BlockPoolSlice(bpid, this, bpdir, conf);
bpSlices.put(bpid, bp);
}
```

BlockPoolSlice 又是什么类呢，其源码上的注释介绍如下：

```
// 一个 BlockPoolSlice 代表一个 BlockPool 存在于一块磁盘上的一部分，所有同属一个
// BlockPool id 下的 BlockPoolSlice 构成了整个 BlockPool。
class BlockPoolSlice {
...
}
```

注释大意为 BlockPoolSlice 是一个 BlockPool 存储在每块盘上的一部分。比如我们有 3 块盘 A、B、C，每个盘下都有对应目录来存放此 BlockPool id 对应的数据。BlockPool id 又是哪里确定的呢？这里要单独介绍这方面的知识。

DataNode 与 FsVolume 磁盘的相关逻辑关系为：首先是 BlockPool id，这个 id 是在集群第一次开始创建的时候确定的，就是在 NameNode 做 format 操作的时候。不管在后续的集群升级或者搬迁的过程中，这个 id 都不会发生改变。此 id 信息保存在了 NamespaceInfo 这个类中，下面是这个类的介绍说明：

```
// NamespaceInfo 信息在 NameNode 与 DataNode 握手通信的时候将被返回
public class NamespaceInfo extends StorageInfo {
    final String buildVersion;
    // BlockPool 唯一 id
    String blockPoolID = "";
    String softwareVersion;
    long capabilities;
...
}
```

DataNode 与 FsVolume 的关系结构见图 9-1。

回到本节所讨论的慢启动问题，这些操作目前看起来并不耗时。我们继续往里看，进入 BlockPoolSlice 的构造方法：

```
BlockPoolSlice(String bpid, FsVolumeImpl volume, File bpDir,
    Configuration conf) throws IOException {
...
// 新建 DU 操作对象用来加载 DfsUsed 值，如果 DfsUsed 缓存值不生效，
// 则会阻塞地执行 du 命令直到此操作的完成
this.dfsUsage = new DU(bpDir, conf, loadDfsUsed());
this.dfsUsage.start();

// 如果进程突然停止，保证 DfsUsed 值还是能被保存下来
ShutdownHookManager.get().addShutdownHook(
    new Runnable() {
        @Override
        public void run() {
            if (!dfsUsedSaved) {
```

```

        saveDfsUsed();
    }
}
}, SHUTDOWN_HOOK_PRIORITY);
}

```

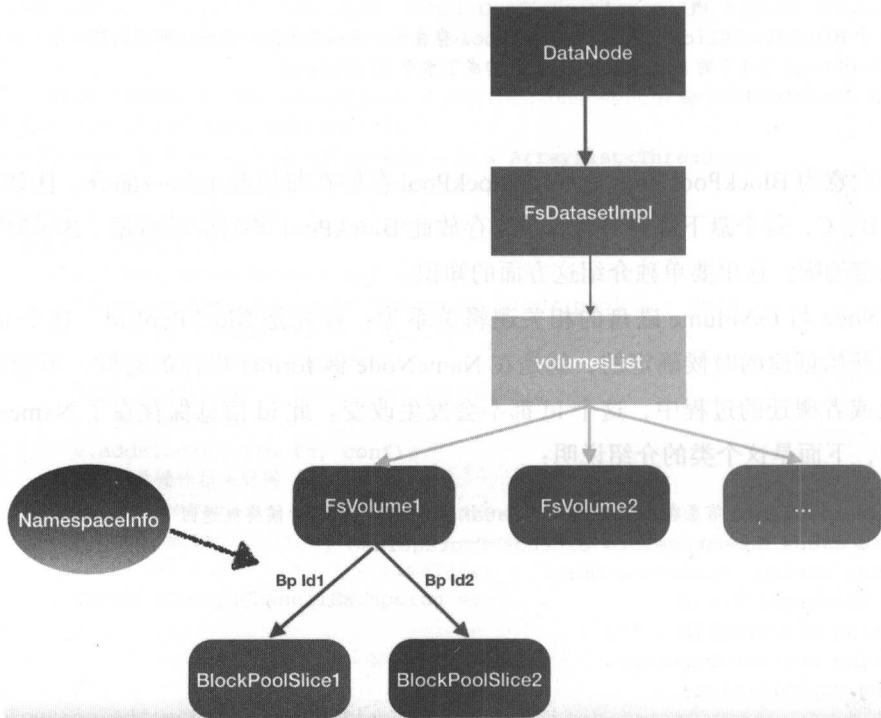


图 9-1 DataNode 与 FsVolume 逻辑关系图

终于在这里我们找到了关键操作，在构造函数中会进行 1 次 du(统计数据使用量)操作。同时注释中特意说明，如果 DfsUsed 缓存值不可用的话，会进行 du 命令操作，阻塞后续操作直到 du 操作完成。那基本我们可以判断出 DataNode“慢启动”场景属于缓存值失效的情况，加载 DfsUsed 缓存值的方法是 loadDfsUsed，进入此方法：

```

long loadDfsUsed() {
    long cachedDfsUsed;
    long mtime;
    Scanner sc;

    try {
        // 初始化文件扫描对象
        sc = new Scanner(new File(currentDir, DU_CACHE_FILE), "UTF-8");
    } catch (FileNotFoundException fnfe) {
        return -1;
    }
}

```

```

try {
    // 判断这个文件中是否有下一个值
    if (sc.hasNextLong()) {
        // 从文件中获取 dfsUsed 缓存值
        cachedDfsUsed = sc.nextLong();
    } else {
        return -1;
    }
    // 判断这个文件中是否有下一个值
    if (sc.hasNextLong()) {
        // 获取文件中的上次记录时间
        mtime = sc.nextLong();
    } else {
        return -1;
    }

    // 如果上次记录时间与当前时间的差值不超过 10 分钟，则可以直接返回上次的缓存值
    // 否则将会返回 -1，从而导致 DataNode 重新对此盘进行 du 计算
    if (mtime > 0 && (Time.now() - mtime < 600000L)) {
        FsDatasetImpl.LOG.info("Cached dfsUsed found for " + currentDir + ": " +
            cachedDfsUsed);
        return cachedDfsUsed;
    }
    return -1;
} finally {
    sc.close();
}
}

```

在这里，会首先从传入的目录地址中读取缓存值文件，如果更新时间在 600 秒以内，则直接读取该值，否则返回 -1，然后重新进行 du 计算。因为笔者的使用场景，使 DataNode 停止服务的时间超过了 30 分钟或者更久，导致更新此缓存文件的间隔时间超过了 600 秒，缓存文件将直接失效，后面的 du 操作自然就发生了。到了这里问题基本锁定了，就是在这个写死的 600 秒上。专业上的术语叫做“hard code”，对于一些标准的变量，这是不合理的，我们要将其可配置化，来适应使用者的需求。至少在笔者的使用情况下，我根本不需要进行 DfsUsed 值的再计算，因为 DataNode 上面的数据没有经过任何的改变。

9.1.3 参数可配置化改造

现在目标已经非常明确了，就是将硬编码值变成可配置化的值，将此配置作为 hdfs-site.xml 的一个新配置项。下面简要说明改造的步骤，首先在 DfsConfigKeys 类里添加 key 名称和默认值：

```

public static final String DFS_DATANODE_CACHED_DFSUSED_CHECK_INTERVAL_MS =
    "dfs.datanode.cached-dfsused.check.interval.ms";

```

```
public static final long DFS_DATANODE_CACHED_DFSUSED_CHECK_INTERVAL_DEFAULT_MS =
    600000;
```

在 `BlockPoolSlice` 中新增变量，并获取此新配置项属性，如果没有设置，则取默认值：

```
class BlockPoolSlice {
    ...
    private final long cachedDfsUsedCheckTime;

    // 在 Federation 使用场景下的扩展问题，每个 BlockPoolSlic 一个 DU 线程，会造成 DU 线程过多的问题
    private final DU dfsUsage;

    BlockPoolSlice(String bpid, FsVolumeImpl volume, File bpDir,
        Configuration conf) throws IOException {
        ...
        // 获取新配置中的缓存检测时间
        this.cachedDfsUsedCheckTime =
            conf.getLong(
                DFSConfigKeys.DFS_DATANODE_CACHED_DFSUSED_CHECK_INTERVAL_MS,
                DFSConfigKeys.DFS_DATANODE_CACHED_DFSUSED_CHECK_INTERVAL_DEFAULT_MS);
    }
}
```

然后在刚刚的 `loadDfsUsed` 方法中替换硬编码值 `600000L`，并更新方法的注释：

```
long loadDfsUsed() {
    ...
    // 将 600000 值替换为变量 cachedDfsUsedCheckTime
    if (mtime > 0 && (Time.now() - mtime < cachedDfsUsedCheckTime)) {
        FsDatasetImpl.LOG.info("Cached dfsUsed found for " + currentDir + ": " +
            cachedDfsUsed);
        return cachedDfsUsed;
    }
    return -1;
} finally {
    sc.close();
}
}
```

最后在 Hadoop 项目中的 `hdfs-default.xml` 文件里加上新的配置项以及描述信息：

```
<property>
  <name>dfs.datanode.cached-dfsused.check.interval.ms</name>
  <value>600000</value>
  <description>DfsUsed 缓存文件的间隔检查时间，默认 10 分钟
</description>
</property>
```

下面编写相应单元测试实例进行测试，单元测试在后面的 [patch](#) 链接中会给出。在 `DataNode` 节点上重新部署新的 jar 包。重启 `DataNode`，重新进行慢启动场景测试，出现下面这个记录，说明是使用了 `DfsUsed` 缓存值，启动过程非常快。

```

2016-01-13 09:13:20,639 INFO [Thread-68] (FsVolumeList.java:402) - Scanning
block pool BP-1543590671-xx.xx.xx.xx-1449897014835 on volume /home/data/data/
hadoop/dfs/data/data9/current...
2016-01-13 09:13:20,639 INFO [Thread-69] (FsVolumeList.java:402) - Scanning
block pool BP-1543590671-xx.xx.xx.xx-1449897014835 on volume /home/data/data/
hadoop/dfs/data/data10/current...
2016-01-13 09:13:20,639 INFO [Thread-71] (FsVolumeList.java:402) - Scanning
block pool BP-1543590671-xx.xx.xx.xx-1449897014835 on volume /home/data/data/
hadoop/dfs/data/data12/current...
2016-01-13 09:13:20,639 INFO [Thread-70] (FsVolumeList.java:402) - Scanning
block pool BP-1543590671-xx.xx.xx.xx-1449897014835 on volume /home/data/data/
hadoop/dfs/data/data11/current...
2016-01-13 09:13:20,822 INFO [Thread-62] (BlockPoolSlice.java:221) - Cached
dfsUsed found for /home/data/data/hadoop/dfs/data/data3/current/BP-1543590671-
xx.xx.xx.xx-1449897014835/current: 40487047190
2016-01-13 09:13:20,825 INFO [Thread-63] (BlockPoolSlice.java:221) - Cached
dfsUsed found for /home/data/data/hadoop/dfs/data/data4/current/BP-1543590671-
xx.xx.xx.xx-1449897014835/current: 39336478590
2016-01-13 09:13:20,825 INFO [Thread-62] (FsVolumeList.java:407) - Time taken
to scan block pool BP-1543590671-xx.xx.xx.xx-1449897014835 on /home/data/data/
hadoop/dfs/data/data3/current: 187ms
2016-01-13 09:13:20,826 INFO [Thread-63] (FsVolumeList.java:407) - Time taken
to scan block pool BP-1543590671-xx.xx.xx.xx-1449897014835 on /home/data/data/
hadoop/dfs/data/data4/current: 189ms

```

此新配置功能笔者已提交开源社区，编号为 HDFS-9624，此 JIRA 中的 patch 链接如下：<https://issues.apache.org/jira/secure/attachment/12782424/HDFS-9624.007.patch>

9.2 Hadoop 中止下线操作后大量剩余复制块问题

Hadoop 集群在使用的过程中，经常会伴随着节点的更新替换：老的、旧的节点下线，新的节点上线。在节点下线的过程中，HDFS 为了保持副本数的统一，会对下线节点中的副本做统一的复制，随后我们会看到大量的副本块准备复制。但是此时如果我们对下线节点进行停止下线的操作，这些副本块是否就不用进行复制了呢？本节所要讲述的内容就是基于这个场景。在下文中，将详细讲述节点下线的原理过程，随后给出针对此现象的解决方案。为了进行一定的比较，本节还将举出 Dead Node 重启的例子。

9.2.1 节点下线操作的含义及问题

这里要解释一个略显专业的名词：节点下线。对应的单词是 Decommission，大意就是说将一个节点从集群中移除，不会对集群造成影响。但是有一点很明确，下线操作必然会导致集群失去大约一个节点资源的计算能力。在 Hadoop 中下线操作最重要的还是两个字：数据。如何保证下线节点中的数据能够完全转移到其他机器中，这才是最关键的。所以在

DataNode 的下线过程中，做的主要操作还是块的重新复制，来达到 HDFS 默认的二副本数据备份的策略。当这些数据都拷贝完毕后，DataNode 的节点状态会变为 Decommissioned 状态。这个时候就可以停止 DataNode，彻底将机器关机并从集群中移除。

本节不是主要介绍 DataNode 下线操作，下线操作中出现的才是本节所重点关注的内容。当你将普通机器进行下线操作时，如果没有什么意外情况，过程会比较顺利。但是当你执行完节点下线操作后，有的时候会出现下面两种情形：

- ❑ 你发现这其实是一个误操作，误将 nodeA 节点加入到 dfs.exclude 文件中了。
- ❑ 你受到上级指示，这个节点暂时不进行下线，重新调回正常服务状态。

上述两种情况发生后，你的第一反应就是将节点从 exclude 文件中移掉，并重新执行 dfsadmin -refreshNodes 命令。当然你会很高兴地看到，节点的状态确实重新变为了 In Service 的状态了。但是如果再仔细一点的话，你会发现 NameNode 页面上的 UnderReplicatedBlocks 块的个数并没有减少，依然是中止下线操作前的数值。这些待复制的块基本就是原下线节点上存储的那些块，见图 9-2。

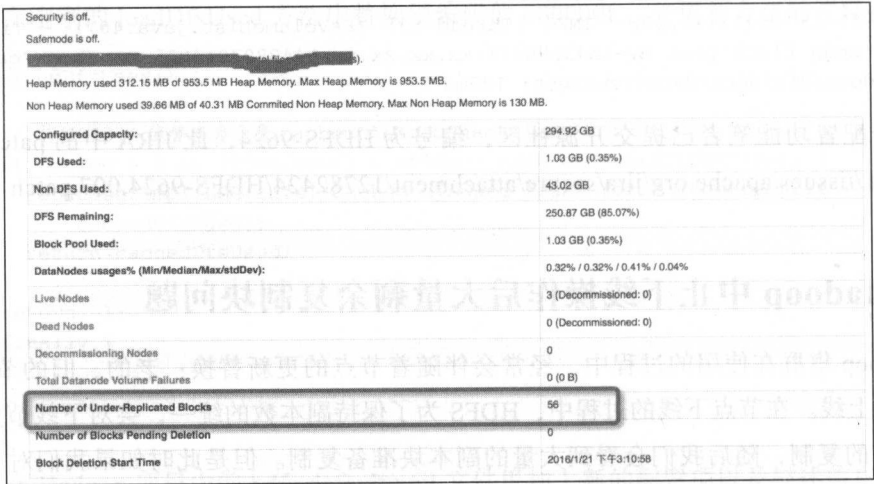


图 9-2 NameNode 的 UnderReplicatedBlocks 计数值

显然当下线节点恢复后，这些大量的复制块是不需要的，因为这会持续占用 NameNode 的时间去处理这些待复制的块。而且在最后 NameNode 会频繁地发现，集群中已经存在足够的块副本了。当有大量的待复制块时，那对 NameNode 来说简直就是灾难，甚至会直接影响到 NameNode 正常的请求处理。从这里可以看出，这不会是一个小问题。在后面的内容中笔者将会介绍一套完整的解决方案。在这之前，为了进行对比，下面再介绍一种类似大量残余复制块的场景。

9.2.2 死节点“复活”

出现大量复制块的另外一个场景是出现死节点 (Dead Node)。当一个 DataNode 长时间不汇报心跳, 超过心跳检测超时时间后, 此 DataNode 就会被认为是 Dead Node。出现了 Dead Node 后, 为了达到副本块的平衡, 同样会进行大量块的拷贝, 与下线操作极为类似。但是这里有一个主要的不同点, 当 Dead Node 重启之后, 这些残余复制块过一会就会减少到 Dead Node 之前的正常值 (大家可以自行执行这个操作进行验证)。两种场景, 相似的现象, 完全不同的结果。Dead Node 恢复的情况才是我们想看到的结果。那么为什么 Dead Node 的恢复会使得复制块减少, 而下线恢复操作则不会呢? 解决这个问题的唯一办法还是得从源码中寻找, 光猜是永远解决不了问题的。一旦发现这个答案, 一定有助于我们用相同的办法解决下线操作时大量复制块残留的问题。

本节内容的焦点始终围绕着“复制块”, 那么在 HDFS 的代码中, 这个变量到底是被哪个对象类所控制呢, 找到这个变量、方法很关键。答案在 FSNamesystem 类中, 代码如下:

```
@Override // FSNamesystemMBean
@Metric
public long getUnderReplicatedBlocks() {
    return blockManager.getUnderReplicatedBlocksCount();
}
```

进一步往里走, 进入 BlockManager 中:

```
public long getUnderReplicatedBlocksCount() {
    return underReplicatedBlocksCount;
}
```

这个变量被谁所赋值的呢, 继续往下看:

```
void updateState() {
    pendingReplicationBlocksCount = pendingReplications.size();
    // 赋值还需要复制的副本数
    underReplicatedBlocksCount = neededReplications.size();
    corruptReplicaBlocksCount = corruptReplicas.size();
}
```

就是中间这行代码, 变量名称表明了它的意思: neededReplications, 需要被复制的副本。我们基本可以推断出, DataNode 在重启之后, 会调用 neededReplication 的移除块类似的操作, 从而使得该变量的大小减少。然后我们再次进行联想, 当 DataNode 进行重启之后, 会首先进行节点的注册动作, 之后会进行心跳的发送, 在发送心跳的时候会进行块的上报, 在块上报的时候显然是一个绝佳的机会。当然这只属于目前的猜想, 我们通过分析代码来验证这个初始猜想。我们进入处理心跳相关的循环方法, BpServiceActor 的 offerService 方法:


```
private void offerService() throws Exception {
    ...
    while (shouldRun()) {
        try {
            final long startTime = scheduler.monotonicNow();
            ...
            // 进行块的上报
            List<DatanodeCommand> cmds = blockReport();
            processCommand(cmds == null ? null : cmds.toArray(new DatanodeCommand[cmds.
                size()]));
            ...
        }
    }
}
```

在操作中，可以看到会有 `blockReport` 的操作，并得到了 `NameNode` 返回给 `DataNode` 的回复命令，进入 `blockReport` 方法：

```
List<DatanodeCommand> blockReport() throws IOException {
    ...

    // 接下来准备发送块报告到 NameNode
    int numReportsSent = 0;
    int numRPCs = 0;
    boolean success = false;
    long brSendStartTime = monotonicNow();
    long reportId = generateUniqueBlockReportId();
    try {
        if (totalBlockCount < dnConf.blockReportSplitThreshold) {
            // 发送块报告到 NameNode
            DatanodeCommand cmd = bpNamenode.blockReport(
                bpRegistration, bpos.getBlockPoolId(), reports,
                new BlockReportContext(1, 0, reportId));
            ...
        }
    }
}
```

在这里，就可以看到 `DataNode` 将块真正汇报给了 `NameNode`，对应到 `NameNode` 的 `RpcServer` 端：

```
@Override // DatanodeProtocol
public DatanodeCommand blockReport(DatanodeRegistration nodeReg,
    String poolId, StorageBlockReport[] reports,
    BlockReportContext context) throws IOException {
    checkNNStartup();
    verifyRequest(nodeReg);
    ...
    // BlockManager 进行上报块的处理
    noStaleStorages = bm.processReport(nodeReg, reports[r].getStorage(),
        blocks, context, (r == reports.length - 1));
    metrics.incrStorageBlockReportOps();
}
...
}
```


在 BlockManager 的 processBlock 方法内还有几层方法，这里给出最后会调用移除复制块动作的方法，如下所示：

```
private Block addStoredBlock(final BlockInfoContiguous block,
                             DatanodeStorageInfo storageInfo,
                             DatanodeDescriptor delNodeHint,
                             boolean logEveryBlock)
throws IOException {
    ...
    // 获取此块期望达到的副本数
    short fileReplication = bc.getBlockReplication();
    // 判断是否还需要再复制副本
    if (!isNeededReplication(storedBlock, fileReplication, numCurrentReplica)) {
        // 不需要了则进行移除
        neededReplications.remove(storedBlock, numCurrentReplica,
                                   num.decommissionedReplicas(), fileReplication);
    } else {
        ...
    }
}
```

在这里会重新进行副本块的判断，如果不需要副本了，则会从 neededReplications 对象中进行删除，因此才会出现待复制块减少的现象。其实还是多亏了 DataNode 重新注册的动作，把自身的所有块重新上报给了 NameNode。而下线节点从下线状态变为正常服务状态，节点不会进行重新注册的动作，原始的块没有被修改过也是不会上报的，才有了以上两种截然不同的结果。

9.2.3 Decommission 下线操作如何运作

这里是本节的一个分界线，上半部分通过现象找出问题的根源，而下半部分则是学习原理解决问题。要解决 Decommission 下线操作中大量复制块残留的问题，要首先明白它的运行逻辑。我们都知道，下线相关动作是通过 -refreshNodes 命令触发的，对应会执行到 DatanodeManager 的 refreshDatanodes 方法：

```
// 刷新节点方法
private void refreshDatanodes() {
    for(DatanodeDescriptor node : datanodeMap.values()) {
        // 判断是否包含在 include 文件中
        if (!hostFileManager.isIncluded(node)) {
            node.setDisallowed(true);
        } else {
            // 如果此节点在 exclude 文件中，则表明需要下线，进行下线操作
            if (hostFileManager.isExcluded(node)) {
                decomManager.startDecommission(node);
            } else {
                // 否则对此节点进行中止下线操作
                decomManager.stopDecommission(node);
            }
        }
    }
}
```

```

    }
}
}

```

在这里我们关注的方面有 2 个：一个是开始下线操作，另外一个则是中止下线操作。

1. 开始下线

在开始下线操作后，待复制块是如何被加入到 `needReplications` 这个对象里去的呢？如下代码所示：

```

public void startDecommission(DatanodeDescriptor node) {
    if (!node.isDecommissionInProgress()) {
        if (!node.isAlive) {
            LOG.info("Dead node {} is decommissioned immediately.", node);
            node.setDecommissioned();
        } else if (!node.isDecommissioned()) {
            for (DatanodeStorageInfo storage : node.getStorageInfos()) {
                LOG.info("Starting decommission of {} {} with {} blocks",
                    node, storage, storage.numBlocks());
            }
            // 在 HeartbeatManager 中更新此节点的状态信息，此节点将会被标记为下线中的状态
            hbManager.startDecommission(node);
            node.decommissioningStatus.setStartTime(monotonicNow());
            // 将目标下线节点加入 pendingNodes 列表中
            pendingNodes.add(node);
        }
    }
    ...
}

```

在代码的最后一行，这个节点被加入到了 `pendingNodes` 列表中。如果各位同学之前研究过 `DecommissionManager` 这个类，应该知道里面会有一个专门的线程用以监视下线中的节点是否已经结束。此监控类代码定义如下：

```

private class Monitor implements Runnable {
    ...

    @Override
    public void run() {
        if (!namesystem.isRunning()) {
            LOG.info("Namesystem is not running, skipping decommissioning checks"
                + ".");
            return;
        }
        ...
    }
}

```

在 `run` 方法中，会进行 2 个操作：

```

@Override

```

```

public void run() {
    ...
    try {
        processPendingNodes();
        check();
    } finally {
        namesystem.writeUnlock();
    }
    if (numBlocksChecked + numNodesChecked > 0) {
        LOG.info("Checked {} blocks and {} nodes this tick", numBlocksChecked,
            numNodesChecked);
    }
}

```

`processPendingNodes` 的作用是将之前加入到 `pendingNodes` 对象中的节点逐步移出到下线节点中:

```

private void processPendingNodes() {
    while (!pendingNodes.isEmpty() &&
        (maxConcurrentTrackedNodes == 0 ||
            decomNodeBlocks.size() < maxConcurrentTrackedNodes)) {
        // 将之前加入到 pendingNodes 列表中的待下线节点加入到 decomNodeBlocks 中
        decomNodeBlocks.put(pendingNodes.poll(), null);
    }
}

```

`check` 方法才是真正的块扫描操作, 判断是否还有副本数不足的块:

```

private void check() {
    final Iterator<Map.Entry<DatanodeDescriptor, AbstractList<BlockInfoContiguous>>>
        it = new CyclicIteration<>(decomNodeBlocks, iterkey).iterator();
    final LinkedList<DatanodeDescriptor> toRemove = new LinkedList<>();

    while (it.hasNext())
        && !exceededNumBlocksPerCheck()
        && !exceededNumNodesPerCheck()) {
        ...
        if (blocks == null) {
            // 这是一个新添加的下线节点, 需要进行全盘的扫描来确定哪些块
            // 还需要进行复制
            LOG.debug("Newly-added node {}, doing full scan to find " +
                "insufficiently-replicated blocks.", dn);
            blocks = handleInsufficientlyReplicated(dn);
            decomNodeBlocks.put(dn, blocks);
            fullScan = true;
        } else {
            // 继续检测此节点上是否还有需要再复制的块, 直到上面的块已全部复制完毕, 以此达到副本数的要求
            LOG.debug("Processing decommission-in-progress node {}", dn);
            pruneSufficientlyReplicated(dn, blocks);
        }
    }
}

```

```

    }
    ...

```

check 方法内的判断逻辑比较多，归纳地来说是持续判断是否还存在副本数量不足的块，不够则继续监控，直到这个数值为 0，然后从下线节点中移除。感兴趣的同学，可以自行研究。在 `handleInsufficientlyReplicated` 内部函数的操作中，会将副本数量不足的块加入到 `neededReplication` 中。

```

private void processBlocksForDecomInternal(
    final DatanodeDescriptor datanode,
    final Iterator<BlockInfoContiguous> it,
    final List<BlockInfoContiguous> insufficientlyReplicated,
    boolean pruneSufficientlyReplicated) {
    ...
    // 利用 DataNode 上的迭代器对象，进行块的遍历
    while (it.hasNext()) {
        ...
        // 获取块当前的副本数
        final NumberReplicas num = blockManager.countNodes(block);
        final int liveReplicas = num.liveReplicas();
        final int curReplicas = liveReplicas;

        // 判断此块是否还需要进行副本的复制
        if (blockManager.isNeededReplication(block, bc.getBlockReplication(),
            liveReplicas)) {
            if (!blockManager.neededReplications.contains(block) &&
                blockManager.pendingReplications.getNumReplicas(block) == 0 &&
                namesystem.isPopulatingReplQueues()) {
                // 如果还需要，则会加入到 BlockManager 的 neededReplication 列表中，
                // 等待进行此副本的复制
                blockManager.neededReplications.add(block,
                    curReplicas,
                    num.decommissionedReplicas(),
                    bc.getBlockReplication());
            }
        }
        ...
    }
}

```

这也就是为什么待复制块骤然增加的原因。

2. 中止下线

那中止下线的过程中又做了什么操作呢？至少有一点我们可以确定，它没有将原本存在于下线节点中的块从 `neededReplications` 对象中移除掉。

```

void stopDecommission(DatanodeDescriptor node) {
    if (node.isDecommissionInProgress() || node.isDecommissioned()) {
        LOG.info("Stopping decommissioning of node {}", node);
    }
}

```

```

// HeartbeatManager 更新此节点的信息
hbManager.stopDecommission(node);
// 清除之前下线过程中已经复制的多余的副本块
if (node.isAlive) {
    blockManager.processOverReplicatedBlocksOnReCommission(node);
}
// 在 DecommissionManager 对象中移除此节点
pendingNodes.remove(node);
decomNodeBlocks.remove(node);
} else {
    LOG.trace("stopDecommission: Node {} is not decommission in progress " +
        "or decommissioned, nothing to do.", node);
}
}

```

操作并不多，可以看到这里只是将下线节点移除，还做了多余副本块的清除，这些副本块是之前下线状态中复制的那些副本块。的确少了我们所需要的那个移除动作，所以我们需要在 `processOverReplicatedBlocksOnReCommission` 方法之后增加这样的处理方法。

9.2.4 中止下线操作后移除残余副本块解决方案

我们首先在 `DecommissionManager` 这个类中定义一个新的方法，如下：

```

// 对指定节点判断其上的块是否还需要复制副本
private void removeNeededReplicatedBlocksInDecomNodes(
    final DatanodeDescriptor datanode) {
    // 获取块迭代器
    final Iterator<BlockInfoContiguous> it = datanode.getBlockIterator();
    // 遍历此节点上的块
    while (it.hasNext()) {
        final BlockInfoContiguous block = it.next();
        BlockCollection bc = blockManager.blocksMap.getBlockCollection(block);
        if (bc == null) {
            // Orphan block, will be invalidated eventually. Skip.
            continue;
        }

        final NumberReplicas num = blockManager.countNodes(block);
        final int liveReplicas = num.liveReplicas();
        final int curReplicas = liveReplicas;
        // 判断是否需要复制此副本
        if (!blockManager.isNeededReplication(block, bc.getBlockReplication(),
            liveReplicas)) {
            // 如果不需要，则进行移除
            blockManager.neededReplications.remove(block, curReplicas,
                num.decommissionedReplicas(), bc.getBlockReplication());
        }
    }
}

```

逻辑很简单，判断是否还需要副本，如果不需要则进行移除。而传入的节点就是原下线节点，有点重新注册的意思在里面。然后加入到 `stopDecommission` 操作中，修改后的代码如下：

```
void stopDecommission(DatanodeDescriptor node) {
    if (node.isDecommissionInProgress() || node.isDecommissioned()) {
        LOG.info("Stopping decommissioning of node {}", node);
        // 获取节点之前的下线状态
        AdminStates adminState = node.getAdminState();
        // HeartbeatManager 更新此节点的信息
        hbManager.stopDecommission(node);
        // 之前下线过程中已复制的多余副本块将会被清除
        if (node.isAlive()) {
            blockManager.processOverReplicatedBlocksOnReCommission(node);
            // 在此新增移除块方法，只有当节点之前的状态为下线中时才有必要执行
            if (adminState == AdminStates.DECOMMISSION_INPROGRESS) {
                removeNeededReplicatedBlocksInDecomNodes(node);
            }
        }
        // 在 DecommissionManager 中移除当前传入的节点
        pendingNodes.remove(node);
        decomNodeBlocks.remove(node);
    } else {
        LOG.trace("stopDecommission: Node {} is not decommission in progress " +
            "or decommissioned, nothing to do.", node);
    }
}
```

这里需要判断一下节点状态，因为如果节点已经是 `Decommissioned` 状态，那么表明待复制块基本已经被复制完了，这个操作意义不大。下面附上单元测试代码，测试已通过。

```
@Test
public void testDecommissionRemovingNeededReplicatedBlocks()
    throws IOException, InterruptedException {
    int underReplicatedBlocksNum;
    int neededReplicatedBlocksNum;
    int sleepIntervalTime = 5000;
    int numNamenodes = 1;
    int numDatanodes = 2;
    // 设置块副本数为集群中的 DataNode 数量，借此让每个节点恰好包含一个
    // 文件所包含的所有块
    int replicas = numDatanodes;
    conf.setInt(DFSConfigKeys.DFS_REPLICATION_KEY, replicas);
    startCluster(numNamenodes, numDatanodes, conf);

    ArrayList<ArrayList<DatanodeInfo>> namenodeDecomList =
        new ArrayList<ArrayList<DatanodeInfo>>(numNamenodes);
    for (int i = 0; i < numNamenodes; i++) {
        namenodeDecomList.add(i, new ArrayList<DatanodeInfo>(numDatanodes));
```

```

}

// 计算每个文件需要存储的块总数
neededReplicatedBlocksNum = (int) Math.ceil(1.0 * fileSize / blockSize);
Path file = new Path("testDecommission.dat");
for (int iteration = 0; iteration < numDatanodes - 1; iteration++) {
    // 准备开始下线操作
    for (int i = 0; i < numNamenodes; i++) {
        FileSystem fileSys = cluster.getFileSystem(i);
        FSNamesystem ns = cluster.getNamesystem(i);
        BlockManager blkocManager = ns.getBlockManager();
        // 写入文件到集群
        writeFile(fileSys, file, replicas);

        DFSClient client = getDfsClient(cluster.getNameNode(i), conf);
        DatanodeInfo[] info = client.datanodeReport(DatanodeReportType.LIVE);

        ArrayList<String> decommissionedNodes = new ArrayList<String>();
        decommissionedNodes.add(info[0].getXferAddr());
        // 写入待下线节点到 exclude 文件
        writeConfigFile(excludeFile, decommissionedNodes);
        // 执行刷新节点方法, 开始下线目标节点
        refreshNodes(cluster.getNamesystem(i), conf);
        // Return the datanode descriptor for the given datanode.
        NameNodeAdapter.getDatanode(cluster.getNamesystem(i), info[0]);

        // 睡眠等待一段时间让 DecommissionManager 内的监控线程去扫描
        // 下线中的块
        Thread.sleep(sleepIntervalTime);
        underReplicatedBlocksNum =
            blkocManager.getUnderReplicatedNotMissingBlocks();
        // 下线节点中待复制的副本数应该等于一个文件所包含的总块数
        assertEquals(neededReplicatedBlocksNum, underReplicatedBlocksNum);

        // 清空下线节点
        decommissionedNodes.clear();
        // 将空节点写入 exclude 配置, 表明此时准备中止下线操作
        writeConfigFile(excludeFile, decommissionedNodes);
        // 重新执行刷新节点操作
        refreshNodes(cluster.getNamesystem(i), conf);

        // 重新获取待复制节点数
        underReplicatedBlocksNum =
            blkocManager.getUnderReplicatedNotMissingBlocks();
        // 中止下线操作后, 待复制副本数理应被清空, 数值应为 0
        assertEquals(0, underReplicatedBlocksNum);

        cleanupFile(fileSys, file);
    }
}

```

```
// 测试结束, 关闭集群
cluster.shutdown();
startCluster(numNamenodes, numDatanodes, conf);
cluster.shutdown();
}
```

解决方案就是以上的几十行代码,但是要写出上述的几十行代码,需要我们对 HDFS 下线机制以及周边原理的了解,其实并没有想象得那么简单。希望大家有所收获,针对这个问题,同样地笔者已提交开源社区,编号为 HDFS-9685。

9.3 DFSOutputStream 的 DataStreamer 线程泄漏问题

DFSOutputStream 类是 HDFS 中的数据写出类,此类控制着 HDFS 数据块的写出操作。在 DFSOutputStream 类内部,通过 DataStreamer、ResponseProcessor 对象之间的合作,最终完成了数据包的传输与写出。但是在程序运行的过程中,DataStreamer 存在部分线程泄漏的问题。本节将首先为大家介绍 DFSOutputStream 内部的数据处理过程,然后在本节末尾对 DataStreamer 的线程泄漏问题进行一定地分析并给出解决方案。

9.3.1 DFSOutputStream 写数据过程及周边相关类、变量

本节主要讲述的是 DataNode 写数据的过程,在写数据的过程中,第一个联想到的就是 DFSOutputStream 对象类。但其实这只是其中的一个大类,它的内部还包括了数据写出的许多类对象。下面主要介绍这几个类。

1. DataStreamer

数据流类,这是数据写操作时调用的主要类。DFSOutputStream 的 start 方法调用的就是 DataStreamer 线程 run 方法。DFSOutputStream 的主要操作都是依靠内部对象类 DataStreamer 来实现的,可以说二者的联系最为紧密。

2. ResponseProcessor

ResponseProcessor 类是 DataStreamer 中的内部类,主要作用是接收 Pipeline 中 DataNode 的 ACK 回复。它是一个线程类,以下为源码中的注释:

处理下游 DataNode 返回的回复信息。当有新的回复抵达的时候,数据包将会从 ACK 队列中移除。

3. DFSPacket

数据包类,在 DataStreamer 和 DFSOutputStream 中都是用这个类进行数据的传输,以下为源码中的注释:

DFSPacket 对象被 DataStreamer 和 DFSOutputStream 所共同使用。DFSOutputStream 产生这些数据包对象, 然后让 DataStreamer 对象发送这些数据包到 DataNode 上。

除了以上 3 个大类需要了解之外, 还有几个变量同样需要重视, 因为这些变量会在后面的分析中经常出现:

- ❑ dataQueue (List<DFSPacket>): 待发送数据包列表。
- ❑ ackQueue (List<DFSPacket>): 数据包回复列表。数据包发送成功后, DFSPacket 将会从 dataQueue 移到 ackQueue 中。
- ❑ Pipeline: Pipeline 是一个常见的名词, 中文翻译的意思是“管道”, 但是笔者认为对此更好的理解方式是“流水线模型”。Pipeline 中的 DataNode 拥有它上游的节点以及下游的节点。

9.3.2 DataStreamer 数据流对象

要了解写数据的细节, 需要先了解 DataStreamer 的实现机理, 因为 DFSOutputStream 的主操作无非是调用了 DataStreamer 的内部方法。DataStreamer 源码中的注释很好地解释了 DataStreamer 所要做的事, 下面是对其简要的概述。

DataStreamer 对象类主要负责发送数据包到 Pipeline 的各个 DataNode 中。它会从 NameNode 中寻求一个新的块 Id 和块的位置信息, 然后开始以流式的方式对 Pipeline 中的 DataNode 进行数据包的传输。每个包有属于它自己的一个数字序列号。当属于一个块的所有的数据包发送完毕并且对应的 ACK 回复都被接收到了, 则表明此次的块写入完成, DataStreamer 将会关闭当前块。DataStreamer 线程从 dataQueue 中选取数据包, 发送此数据包给 Pipeline 中的首个 DataNode。然后移动此数据从 dataQueue 列表到 ackQueue。ResponseProcessor 会从各个 DataNode 中接收 ACK 回复。

对于每一个发送的数据包而言, 只有当 Pipeline 中的 DataNode 都发送了成功的 ACK 回复, 才表明此数据包已成功写入节点。然后 ResponseProcessor 将会从 ackQueue 列表中移除相应的数据包。当出现错误的时候, 所有未完成的数据包将会从 ackQueue 中移除掉。会重新建立一个新的 Pipeline, 移除掉坏的 DataNode, 然后 DataStreamer 会从 dataQueue 中重新发送此数据包。

总体过程大致如此, 想必大家或多或少已经对其中的过程有所了解。图 9-3 为 DataStreamer 数据流的相关结构。

图 9-3 对应的程序逻辑在 run 方法中。首先在 while 循环中会获取一个数据包:

```
one = dataQueue.getFirst();
```

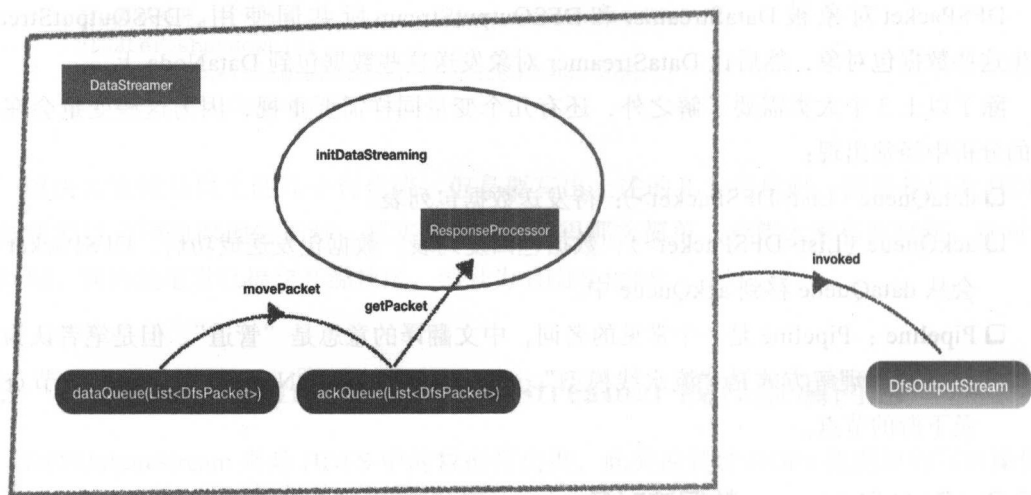


图 9-3 DataStreamer 数据流相关结构图

在接下来的操作中会出现数据包的转移：

```
// 下面开始发送数据包
SpanId spanId = SpanId.INVALID;
synchronized (dataQueue) {
    // 移动数据包从 dataQueue 队列到 ackQueue 队列
    if (!one.isHeartbeatPacket()) {
        if (scope != null) {
            spanId = scope.getSpanId();
            scope.detach();
            one.setTraceScope(scope);
        }
        scope = null;
        dataQueue.removeFirst();
        ackQueue.addLast(one);
        dataQueue.notifyAll();
    }
}
```

然后发送数据到远程 DataNode 节点：

```
// 写出数据到远程 DataNode 节点
try (TraceScope ignored = dfsClient.getTracer().
    newScope("DataStreamer#writeTo", spanId)) {
    one.writeTo(blockStream);
    blockStream.flush();
} catch (IOException e) {
    ...
}
```

dataStreamer 发送完数据包之后，responseProcessor 进程会收到来自 DataNode 的 ACK 回复。如果对于一个块，收到了 Pipeline 中 DataNode 所有的 ACK 回复信息，则代表这个

块发送完成了。Pipeline 的 DataNode 构建分为两种情况，代表着两种情形的数据传输：

□ BlockConstructionStage.PIPELINE_SETUP_CREATE

□ BlockConstructionStage.PIPELINE_SETUP_APPEND

第一种情况，在新分配块的时候进行的。从 NameNode 上获取新的块 Id 和位置，然后连接上第一个 DataNode。

```
if (stage == BlockConstructionStage.PIPELINE_SETUP_CREATE) {
    if (LOG.isDebugEnabled()) {
        LOG.debug("Allocating new block: " + this);
    }

    setPipeline(nextBlockOutputStream());
    initDataStreaming();
}
```

nextBlockOutputStream 方法执行如下，在此方法中将会连接上第一个 DataNode 节点：

```
protected LocatedBlock nextBlockOutputStream() throws IOException {
    ...

    // 连接第一个 DataNode
    success = createBlockOutputStream(nodes, storageTypes, 0L, false);
    ...
}
```

Pipeline 的第一阶段如图 9-4 所示。

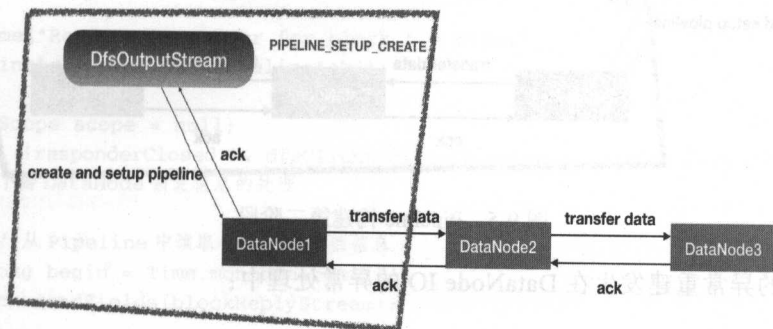


图 9-4 Pipeline 构建第一阶段

另外一个阶段是第一个 DataNode 节点向其他剩余节点建立连接：

```
} else if (stage == BlockConstructionStage.PIPELINE_SETUP_APPEND) {
    if (LOG.isDebugEnabled()) {
        LOG.debug("Append to block {}", block);
    }
    ...

    setupPipelineForAppendOrRecovery();
}
```

```

    if (streamerClosed) {
        continue;
    }
    initDataStreaming();
}

```

后面是建立连接的代码：

```

private void setupPipelineForAppendOrRecovery() throws IOException {
    ...
    setupPipelineInternal(nodes, storageTypes);
}

protected void setupPipelineInternal(DatanodeInfo[] datanodes,
    StorageType[] nodeStorageTypes) throws IOException {
    ...

    // 建立剩余节点的连接
    success = createBlockOutputStream(nodes, storageTypes, newGS, isRecovery);
    ...
}

```

第二阶段过程如图 9-5 所示。

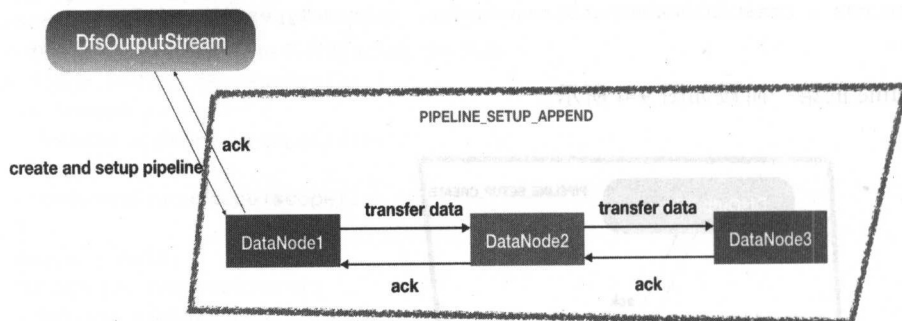


图 9-5 Pipeline 构建第二阶段

Pipeline 的异常重建发生在 DataNode IO 的异常处理中：

```

public void run() {
    long lastPacket = Time.monotonicNow();
    TraceScope scope = null;
    while (!streamerClosed && dfsClient.clientRunning) {
        ...

        DFSPacket one;
        try {
            // IO 错误处理
            boolean doSleep = processDatanodeOrExternalError();
            ...
        }
    }
}

```

```

// 数据流传输出错处理
private boolean processDatanodeOrExternalError() throws IOException {
// 如果没有发生出错情况,则返回
if (!errorState.hasDatanodeError() && !shouldHandleExternalError()) {
return false;
}

...

if (response != null) {
LOG.info("Error Recovery for " + block +
" waiting for responder to exit. ");
return true;
}
// 关闭当前 stream 对象
closeStream();

...
// Pipeline 错误恢复处理
setupPipelineForAppendOrRecovery();
...

```

9.3.3 ResponseProcessor 回复获取类

进入 ResponseProcessor 类的主运行方法:

```

public void run() {

setName("ResponseProcessor for block " + block);
PipelineAck ack = new PipelineAck();

TraceScope scope = null;
while (!responderClosed && dfsClient.clientRunning && !isLastPacketInBlock) {
// 下游 DataNode 回复信息的处理
try {
// 从 Pipeline 中读取一条 ACK 返回信息
long begin = Time.monotonicNow();
ack.readFields(blockReplyStream);
...

```

这里会从 blockReplyStream 输入流中读取 ACK 返回信息,要特别注意的是,这里读到的 ACK 与之前 ackQueue 中的 ACK 并不是同一个对象。这个 ACK 指的是 PipelineAck,主要的作用是获取其中的 seqno 序列号。

```
long seqno = ack.getSeqno();
```

判断是否是有效的块回复:

```
assert seqno != PipelineAck.UNKOWN_SEQNO :
```

```

        "Ack for unknown seqno should be a failed ack: " + ack;
// 如果是心跳回复信息, 则跳过此轮处理
if (seqno == DFSPacket.HEART_BEAT_SEQNO) {
    continue;
}

```

然后取出 ACK 的 DFSPacket 数据包, 比较序列号, 判断是否一致:

```

// 获取一个数据包的 ACK 信息
DFSPacket one;
synchronized (dataQueue) {
    one = ackQueue.getFirst();
}
if (one.getSeqno() != seqno) {
    throw new IOException("ResponseProcessor: Expecting seqno " +
        " for block " + block +
        one.getSeqno() + " but received " + seqno);
}

```

此 ACK 回复包判断完毕后, 会进行相应数据包的移除:

```

synchronized (dataQueue) {
    scope = one.getTraceScope();
    if (scope != null) {
        scope.reattach();
        one.setTraceScope(null);
    }
    lastAackedSeqno = seqno;
    pipelineRecoveryCount = 0;
    ackQueue.removeFirst();
    dataQueue.notifyAll();

    one.releaseBuffer(byteArrayManager);
}

```

至此 ackQueue 中的数据包就被彻底移除掉了, 从最开始加入到 dataQueue, 到移动到 ackQueue, 到最后回复的确认完毕, 进行最终的移除。

在这些操作执行期间, 还会进行一项判断:

```
isLastPacketInBlock = one.isLastPacketInBlock();
```

如果此数据包是发送块的最后一个数据包, 则此 responseProcessor 线程将会退出循环:

```
while (!responderClosed && dfsClient.clientRunning && !isLastPacketInBlock)
```

当运行期间发生异常, 会导致 responderClosed 设置为 true, 同样会导致循环的退出:

```

catch (Exception e) {
    if (!responderClosed) {
        lastException.set(e);
        errorState.setInternalError();
    }
}

```

```

errorState.markFirstNodeIfNotMarked();
synchronized (dataQueue) {
    dataQueue.notifyAll();
}
if (!errorState.isRestartingNode()) {
    LOG.warn("Exception for " + block, e);
}
// 发生异常会使得 responderClosed 设置为 true
responderClosed = true;
}

```

ResponseProcessor 内部执行流程见图 9-6。

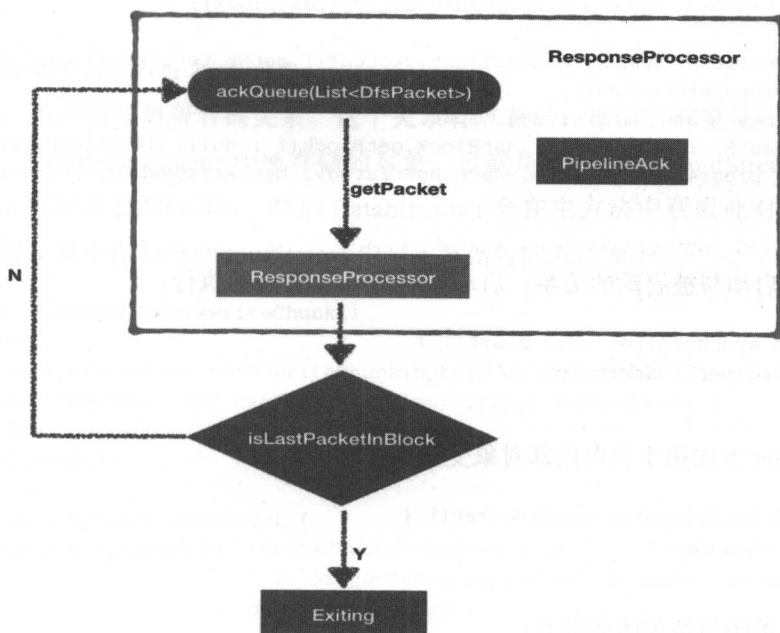


图 9-6 ResponseProcessor 内部执行过程图

9.3.4 DataStreamer 与 DFSOutputStream 的关系

在上文中已经或多或少提到了这两个类之间的关系，可简要概况为以下 4 种关系：

- 创建与被创建的关系。
- 启动与被启动的关系。
- 关闭与被关闭的关系。
- 生产者与消费者的关系。

下面一一做简要的分析。

第一点，创建与被创建的关系，可以从 DFSOutputStream 的构造函数中看出：

```

private DFSOutputStream(DFSClient dfsClient, String src,
    EnumSet<CreateFlag> flags, Progressable progress, LocatedBlock lastBlock,
    HdfsFileStatus stat, DataChecksum checksum, String[] favoredNodes)
    throws IOException {
    ...

    // 以下是 DataStreamer 的构建
    if (!toNewBlock && lastBlock != null) {
        // 如果此块不是一个新块且最后一个块不为空, 则 append 操作是追加在一个存在的块上
        streamer = new DataStreamer(lastBlock, stat, dfsClient, src, progress,
            checksum, cachingStrategy, byteArrayManager);
        getStreamer().setBytesCurBlock(lastBlock.getBlockSize());
        adjustPacketChunkSize(stat);
        getStreamer().setPipelineInConstruction(lastBlock);
    } else {
        computePacketChunkSize(dfsClient.getConf().getWritePacketSize(),
            bytesPerChecksum);
        streamer = new DataStreamer(stat,
            lastBlock != null ? lastBlock.getBlock() : null, dfsClient, src,
            progress, checksum, cachingStrategy, byteArrayManager, favoredNodes);
    }
}

```

第二点, 启动与被启动的关系, 启动指的是 `start` 方法的执行。

```

protected synchronized void start() {
    getStreamer().start();
}

```

`getStreamer` 方法用于获取内部对象变量 `dataStreamer`:

```

protected DataStreamer getStreamer() {
    return streamer;
}

```

第三点, 关闭与被关闭的关系。

```

public void close() throws IOException {
    synchronized (this) {
        try (TraceScope ignored = dfsClient.newPathTraceScope(
            "DFSOutputStream#close", src)) {
            closeImpl();
        }
    }
    dfsClient.endFileLease(fileId);
}

protected synchronized void closeImpl() throws IOException {
    ...
    closeThreads(true);
    ...
}

```



```
// 关闭 dataStreamer 以及 responseProcessor 对象, force 参数表示是否强制关闭
protected void closeThreads(boolean force) throws IOException {
try {
    // Streamer 对象的关闭
    getStreamer().close(force);
    getStreamer().join();
    getStreamer().closeSocket();
} catch (InterruptedException e) {
    throw new IOException("Failed to shutdown streamer");
} finally {
    getStreamer().setSocketToNull();
    setClosed();
}
}
```

在这里会关闭 streamer 相关的类。

第四点, 生产者与消费者的关系。这个关系有点意思, 那消费对象是什么呢? 答案是 DFSPacket, 也就是 dataQueue 中所存储的对象。也就是说, DFSOutputStream 中的方法会往 dataQueue 中存入 DFSPacket, 然后 DataStreamer 会在主方法中获取此对象, 也就是上文分析的场景。其中在 DFSOutputStream 中写入数据包的方法如下:

```
// @see FSOutputSummer#writeChunk()
@Override
protected synchronized void writeChunk(byte[] b, int offset, int len,
    byte[] checksum, int ckoff, int cklen) throws IOException {
    dfsClient.checkOpen();
    checkClosed();

    if (len > bytesPerChecksum) {
        throw new IOException("writeChunk() buffer size is " + len +
            " is larger than supported bytesPerChecksum " +
            bytesPerChecksum);
    }
    if (cklen != 0 && cklen != getChecksumSize()) {
        throw new IOException("writeChunk() checksum size is supposed to be " +
            getChecksumSize() + " but found to be " + cklen);
    }
    // 如果当前数据包为空, 则进行构造
    if (currentPacket == null) {
        currentPacket = createPacket(packetSize, chunksPerPacket, getStreamer()
            .getBytesCurBlock(), getStreamer().getAndIncCurrentSeqno(), false);
        ...
    }
    // 写入数据到当前的数据包中
    currentPacket.writeChecksum(checksum, ckoff, cklen);
    currentPacket.writeData(b, offset, len);
    currentPacket.incNumChunks();
    getStreamer().incBytesCurBlock(len);
}
```

```
// 如果当前数据包已经写满, 则加入到队列中等待传输
if (currentPacket.getNumChunks() == currentPacket.getMaxChunks() ||
    getStreamer().getBytesCurBlock() == blockSize) {
    enqueueCurrentPacketFull();
}
}
```

`enqueueCurrentPacketFull` 方法会将数据包写入 `dataQueue` 中:

```
void enqueueCurrentPacket() throws IOException {
    getStreamer().waitAndQueuePacket(currentPacket);
    currentPacket = null;
}
```

在 `DFSOutputStream` 的 `close` 方法中, 也会触发一次最后清洗数据的动作, 将数据写出到各个 `DataNode` 中, 也会调用到 `enqueueCurrentPacket` 方法:

```
protected synchronized void closeImpl() throws IOException {
    // 将缓存中的数据进行写出
    flushBuffer();
    // 最后关闭操作的时候, 同样会将数据包加入到队列中
    if (currentPacket != null) {
        enqueueCurrentPacket();
    }

    if (getStreamer().getBytesCurBlock() != 0) {
        setCurrentPacketToEmpty();
    }
    // 最后将所有数据刷出到 DataNode 中
    flushInternal();
    // 在 streamer 对象关闭前获取最后一个块对象
    // 在前面的操作中如果发生异常, 此块对象将会为 null
    lastBlock = getStreamer().getBlock();
    ...
}
```

`DataStreamer` 与 `DFSOutputStream` 之间的关系见图 9-7。

9.3.5 Streamer 线程泄漏问题

`Streamer` 线程泄漏问题是笔者在学习 `DFSOutputStream` 相关原理时发现的, 过程算是比较意外吧。线程泄漏问题可以类比于内存泄漏, 指的是该释放的空间没释放。线程泄漏问题同理, 该关闭的线程对象没有及时关闭。发生的地方自然然是是在 `DFSOutputStream` 的 `close` 方法中, 这里重新调出这段程序:

```
public void close() throws IOException {
    synchronized (this) {
        try (TraceScope ignored = dfsClient.newPathTraceScope(
            "DFSOutputStream#close", src)) {
```

```

// 在 closeImpl 方法中可能发生关闭异常
closeImpl();
}
}
dfsClient.endFileLease(fileId);
}

```

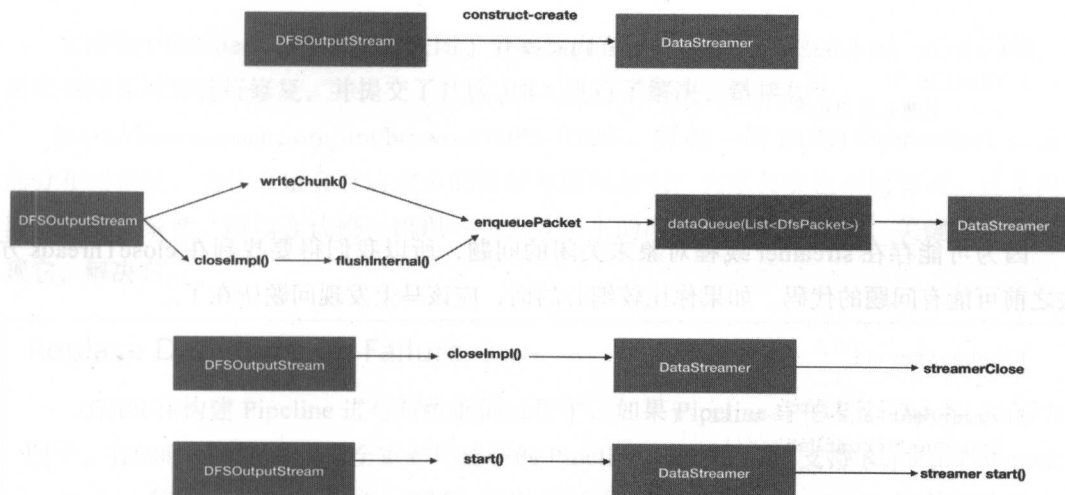


图 9-7 DataStream 与 DFSOutputStream 的关系图

进入 closeImpl 实质的关闭方法，仔细观察每步操作可能存在的问题。

```

protected synchronized void closeImpl() throws IOException {
    if (isClosed()) {
        getStreamer().getLastException().check(true);
        return;
    }
    try {
        // 将缓冲中的数据写出
        flushBuffer();

        if (currentPacket != null) {
            enqueueCurrentPacket();
        }

        if (getStreamer().getBytesCurBlock() != 0) {
            setCurrentPacketToEmpty();
        }
        // 最后将所有数据刷出到 DataNode 中
        flushInternal();
        // 在 streamer 对象关闭前获取最后一个块对象
        // 在前面的操作中如果发生异常，此块对象将会为 null
        ExtendedBlock lastBlock = getStreamer().getBlock();
    }
}

```

```

// 执行线程关闭操作
closeThreads(true);

try (TraceScope ignored =
    dfsClient.getTracer().newScope("completeFile")) {
    // 完成文件操作
    completeFile(lastBlock);
}
} catch (ClosedChannelException ignored) {
} finally {
    // 设置已关闭状态
    setClosed();
}
}

```

因为可能存在 streamer 线程对象未关闭的问题，所以我们得要找到在 closeThreads 方法之前可能有问题的代码。如果你比较细心的话，应该马上发现问题所在了。

```

flushBuffer();

if (currentPacket != null) {
    enqueueCurrentPacket();
}

if (getStreamer().getBytesCurBlock() != 0) {
    setCurrentPacketToEmpty();
}
// 最后将所有的数据刷出到 DataNode 中
flushInternal();

```

从 flushBuffer 到 flushInternal 中的操作都有可能抛出 IO 异常，一旦抛出异常程序将直接跳到 finally 代码处进行处理，中间的 closeThread 方法将不会被执行到，从而导致 DataStreamer 线程泄漏。这个 bug 目前已经提交开源社区，并且已有相应的 patch，编号 HDFS-9812。解决办法很简单，在这层代码中再包一层 try-catch。把 closeThread 方法放入新增 try-catch 方法的末尾进行处理，或者直接将 closeThread 方法直接放入 finally 代码处进行处理，详细信息可以查看 JIRA 链接：

<https://issues.apache.org/jira/browse/HDFS-9812>。

读到这里，如果你仅仅认为只有此处可能会有潜在问题的话，那就错了。在笔者后续对此代码的学习过程中，发现对于 closeImpl 方法，如果发生 IO 异常，还会导致 DFSClient 端无法正常关闭，相关代码如下：

```

public void close() throws IOException {
    synchronized (this) {
        try (TraceScope ignored = dfsClient.newPathTraceScope(
            "DFSOutputStream#close", src)) {

```

```

// 如果 closeImpl 发生 IO 异常，dfsClient.endFileLease 方法则会被跳过
// 从而导致文件没有完全关闭，造成内存泄露
closeImpl();
}
}
dfsClient.endFileLease(fileId);
}

```

文件在 DFSCClient 端没有正常关闭，会导致内存的严重浪费，这绝不是一个小问题。后续笔者同样对其进行修复，并提交了社区 JIRA 进行了解决，链接如下：

<https://issues.apache.org/jira/browse/HDFS-10549>。另起一段 HDFS 作为一套比较成熟的分布式系统，为什么也会有这么多的资源泄露问题呢？大家其实也不用惊讶，正是由于其内部的复杂性，所以在代码上出现一些大大小小的问题还是很正常的，关键是有人能发现它，解决它。

Replace Datanode on Failure

HDFS 在构建 Pipeline 进行写数据的过程中，如果 Pipeline 中的某个 DataNode 写失败了，有的时候并不需要完全重新构建新的 Pipeline。HDFS 可以支持 Replace Datanode on Failure 的策略。也就是说，HDFS 可以重新寻找一个新的 DataNode 来替换失败的 DataNode。但是这需要保证一个前提：HDFS 中有足够的候选节点。如果去除 Pipeline 中的所有 DataNode，集群中并没有多余可选的 DataNode，此策略将不会生效。

9.4 小结

本章所讲述的 3 大异常场景中只有第一种用户会明显感觉到，所以它是一个重点的内容，本章也给出了其中的解决方案，大家可以将其运用到自身的系统中。其余两节内容可能会偏于理论，稍显难懂，建议大家进行源代码的跟踪调试，进一步了解其运行过程。

如何向开源社区提交自己的代码

作为一名职业程序员，如果去除待遇、薪资等等的因素考虑，从纯技术的角度出发，如何才能达到一个比较高的境界呢？答案是与最顶尖的那一批人交流合作。当然最顶尖的牛人几乎都不在身边，而且大多在国外。那么难道就没有办法了吗？不是的，不要忘了还有网络！可以通过社区、邮件进行交流，提出自己的想法。这些人往往活跃于许多开源社区，比如 Apache 社区。所以，如果一个普通开发者能够向开源社区打进自己的补丁 (patch)，并且此部分代码能够被合入主干代码中，将会是令开发者非常骄傲的一件事情。不要以为这个事情很难，开源社区的代码中也可能出现比如显示字符出错这样的低级错误，如果被你发现了，也是一个打补丁的机会。总之一句话，能够向开源社区打补丁的人，也许并不能说明他有多强，但是一定程度上能说明他是具有钻研精神和思考能力的。毕竟大部分开发人员都还只是停留在使用这些开源社区产品的层面上，对于内部复杂的设计与实现却只是略知一二。下面就来讨论一下，作为一个非 Committer，如何向开源社区提交代码，打进自己的补丁。

打补丁的前提

首先打补丁的一个前提条件是你已经拥有分析和改造已有开源系统代码的能力。但是鉴于开源系统本身的复杂性，你可以专注于研究其中部分模块代码。比如 Hadoop 代码，你可以选择研究 HDFS，或者 YARN，亦或者 MapReduce 模块。不管怎么说，你要拥有阅读和改造某块代码的能力。

补丁注意事项

补丁是如何打出的呢？补丁类型的代码在某种程度上指的是被变动过的代码，包括修改代码和增删代码，也就是代码的变化之处。那么有什么工具可以知道代码的变化呢？git diff 命令正好可以帮助我们解决这个问题。通过 `git diff > your desFileName` 命令就可以导出这个补丁文件了，具体的命令使用方法读者可以自行查阅。但是这里又会有一个问题，补丁打出来之后，只能说明这段补丁代码能够运行正常，并不能代表这个补丁遵守了一定的规范。下面列出几个开源社区中对补丁文件的规范要求，从图 A-1 中的 Hadoop QA 测试结果表中，我们能获知一些规范要求。

Vote	Subsystem	Runtime	Comment
-1	pre-patch	19m 11s	Findbugs (version) appears to be broken on trunk.
+1	@author	0m 0s	The patch does not contain any @author tags.
+1	tests included	0m 0s	The patch appears to include 1 new or modified test files.
+1	javac	11m 13s	There were no new javac warning messages.
+1	javadoc	14m 53s	There were no new javadoc warning messages.
+1	release audit	0m 33s	The applied patch does not increase the total number of release audit warnings.
+1	checkstyle	1m 11s	There were no new checkstyle issues.
-1	whitespace	0m 0s	The patch has 1 line(s) that end in whitespace. Use git apply --whitespace=fix.
+1	install	2m 9s	mvn install still works.
+1	eclipse:eclipse	0m 46s	The patch built with eclipse:eclipse.
-1	findbugs	3m 23s	The patch appears to introduce 1 new FindBugs (version 3.0.0) warnings.
+1	native	4m 19s	Pre-build of native portion
-1	hdfs tests	58m 4s	Tests failed in hadoop-hdfs.
		115m 46s	

图 A-1 Hadoop QA 测试结果

主要的规范要求有如下 4 点：

- ❑ 补丁必须是打在最新代码之上的，否则会出现合并代码出错的情况。因为补丁中会有每行代码的文件名和对应位置，如果不是在最新代码上的修改会导致出错的情况。
- ❑ 每行代码的最大长度不能超过 80 个字符，否则会报 checkstyle 错误。所以建议的方法是每次修改完自己的代码后，以 hadoop format 格式文件的方式进行格式化操作。
- ❑ 多余空白行不能有，否则会报 whitespace 警告。多余空白行的意思指的是回车另起一行的时候，一般会有多一行的空白行，要用删除键把此行前面的空格符删去。
- ❑ 补丁代码需要包含对应的测试代码，来证明补丁的可行性。
- ❑ 补丁的一般命名规范是：issue（问题）序号 +.00 提交次序 +.patch。类似如图 A-2 的形式。

Ⓜ HDFS-9343.000.patch	4 kB	2 days ago
Ⓜ HDFS-9343.001.patch	5 kB	2 days ago
Ⓜ HDFS-9343.002.patch	5 kB	2 days ago
Ⓜ HDFS-9343.003.patch	6 kB	2 days ago
Ⓜ HDFS-9343.004.patch	6 kB	Yesterday

图 A-2 补丁命名实例

如何让 Committer 采纳你的补丁

当你把补丁提交到开源社区的时候，一般 Committer 会关注到你的 issue，可能会迟些时间查阅你的补丁。如果他认为不错，会给你提出意见，叫你修改补丁代码，你应该积极响应 Committer 的回复，及时进行补丁的更新，这会让人家对你产生好的印象。如果出现了类似于图 A-3 中 +1 的评语，代表他已经认同你的补丁代码，基本上可以被合入主干代码了。

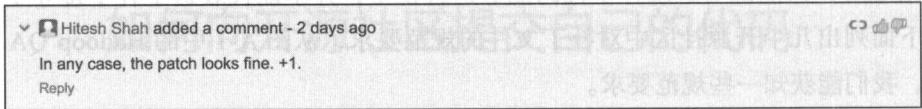


图 A-3 Patch 的回复认同

这就是代码提交的整个流程了，其实并不复杂，但是周期可能会比较长。因为有的时候需要多次修改代码，这需要我们有一定的耐心。

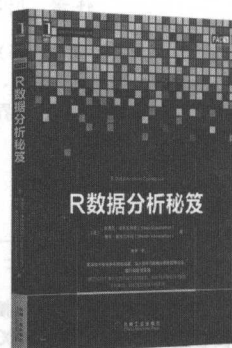
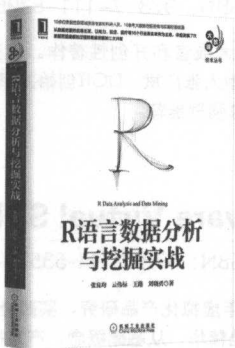
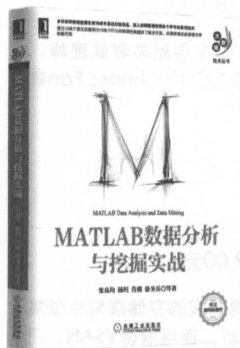
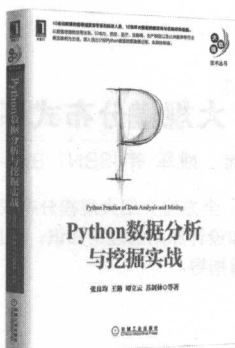
总结

最后希望国内的程序员能够向开源社区多做贡献，提升自身的影响力。下面给出笔者自己早期提交的 2 个 issue 供大家作为参考：

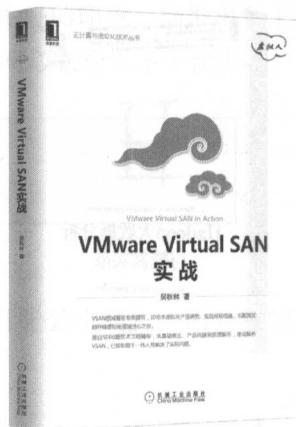
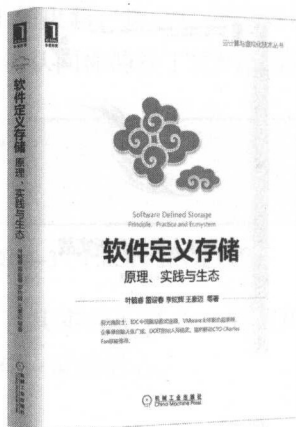
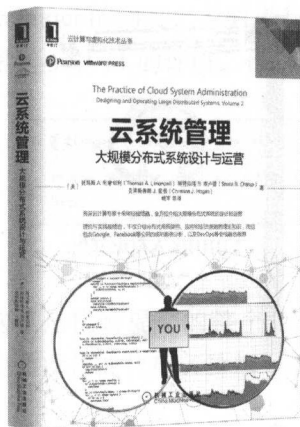
- ❑ <https://issues.apache.org/jira/browse/HDFS-9303>
- ❑ <https://issues.apache.org/jira/browse/MAPREDUCE-6499>

推荐阅读

大数据学习路线图：数据分析与挖掘



推荐阅读



云系统管理：大规模分布式系统设计与运营

作者：托马斯 A. 利蒙切利 等 译者：姚军 等 ISBN: 978-7-111-54160-8 定价：99.00元

资深云计算专家十余年经验结晶，全方位介绍大规模分布式系统的设计和运营；理论与实践相结合，不仅介绍分布式系统架构、应用和设计原则的理论知识，而且包含Google、Facebook等公司的成功案例分析，为系统管理员提供有益指导。

软件定义存储：原理、实践与生态

作者：叶毓睿 等 ISBN: 978-7-111-53957-5 定价：89.00元

软件定义存储（SDS）领域的集大成者和开创性著作。倪光南院士、IDC中国副总裁武连峰、VMware全球副总裁李映、企事录创始人张广斌、DOIT创始人郑信武、猎豹移动CTO Charles Fan等数十位来自学术界和企业界的资深专家强烈推荐。

VMware Virtual SAN实战

作者：吴秋林 ISBN: 978-7-111-53522-5 定价：59.00元

VSAN领域著名专家撰写，10余年虚拟化产品研究、实践经验结晶，名副其实的存储虚拟化领域良心之作。（2）源自5000篇技术文档精华，从基础概念、产品构建到原理解析，逐层解析VSAN，已帮助数千一线人员解决了实际问题。

作者简介

林意群

唯品会上海研发中心数据平台与应用部研发工程师，Apache Hadoop Committer，主要专注于HDFS模块的研究。对大数据处理、分布式计算兴趣浓厚，在实际工作中努力钻研，分享了大量技术文章，贡献了很多实践经验。博客地址为：<http://blog.csdn.net/androidlushangderen>

本书作者是一名Hadoop社区的活跃贡献者，研究并贡献了很多代码。他把工作实践中遇到的许多经验写入了书中，介绍了HDFS未来比较棒的一些功能特性，以及Hadoop社区目前在做的一些事情。在这本书中，你会看到许多与社区相关的JIRA，告诉你如何从社区上找到问题的解决办法。本书适合具有一定Java语言基础的读者。

本书主要内容:

- 缓存管理、快照管理的优势与侧重点。
- HDFS比较新颖的一些功能，以及一些比较少被人使用到的功能特性。
- 流量处理的详细过程，包括HDFS目前流量处理的场景以及Balancer工具的数据平衡原理和优化。
- 多套运维管理的操作方案，包括数据迁移、数据监控等。
- HDFS写磁盘时的一些优化技巧和改造方案。
- HDFS部分发生异常的场景，以及相应的解决办法。



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/大数据

ISBN 978-7-111-56207-8



9 787111 562078 >

定价: 79.00元